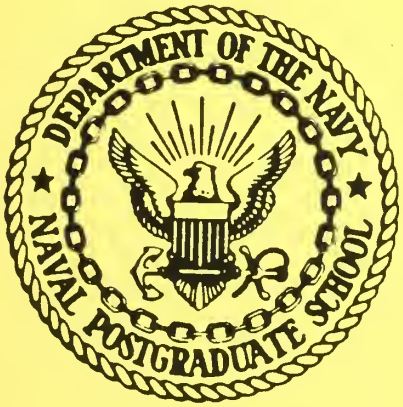
NPS52-88-011

NAVAL POSTGRADUATE SCHOOL
Monterey, California



RULE-BASED MOTION COORDINATION FOR THE ADAPTIVE SUSPENSION VEHICLE

by Sehung Kwak

Robert B. McGhee

May 1988

Prepared for:
Ohio State University
206 W. 18th Avenue
Columbus, Ohio 43210

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. C. Austin
Superintendent

Kneale T. Marshall
Acting Provost

This report is prepared in conjunction with research sponsored in part by contract from the Ohio State University Research Foundation under RF Project No. 716520.

REPORT DOCUMENTATION PAGE

EPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
ECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
ECLASSIFICATION / DOWNGRADING SCHEDULE	
RFORMING ORGANIZATION REPORT NUMBER(S) S52-88-011	5 MONITORING ORGANIZATION REPORT NUMBER(S)
AME OF PERFORMING ORGANIZATION val Postgraduate School — 6b OFFICE SYMBOL (If applicable) Code 52	7a NAME OF MONITORING ORGANIZATION Prof. Kenneth Waldron, Dept. of Mech. Eng. Ohio State University
DDRESS (City, State, and ZIP Code) nterey, CA 93943-5000	7b ADDRESS (City, State, and ZIP Code) 2075 Robinson Laboratory 206 W 18th Avenue Columbus, Ohio 43210
AME OF FUNDING / SPONSORING RGANIZATION Ohio State Univ. Research Foundation — 8b OFFICE SYMBOL (if applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER RF Project No. 716520 RF Purchase Order No. 496549
DDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO

TLE (Include Security Classification)
ULE-BASED MOTION COORDINATION FOR THE ADAPTIVE SUSPENSION VEHICLE (U)

ERSONAL AUTHOR(S) Kwak, S. H. and McGhee, R. B.

TYPE OF REPORT	13b TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
erim Scientific	FROM _____ TO _____	May 1988	129

UPPLEMENTARY NOTATION

COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Robotics, Walking machines, Adaptive Suspension Vehicle, Robot Motion Planning, Rule-based systems

BSTRACT (Continue on reverse if necessary and identify by block number)
is study investigates the utility of rule-based coordination of motion for rough-terrain comotion by a hexapod walking machine. The logic for generating leg commands is written Prolog while the simulation of the terrain and of the vehicle kinematics, as well as low vel on-board computer functions, are written in extended Common Lisp. It is found that is approach results in code that is much easier to understand and modify than previous tion coordination programs written in Pascal. The authors believe that both the method-ogy and the stepping logic presented in this report possess sufficient merit to justify ll-scale physical testing in the Adaptive Suspension Vehicle operated under DARPA contract Ohio State University.

ISTRIBUTION / AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
NAME OF RESPONSIBLE INDIVIDUAL obert B. McGhee	22b TELEPHONE (Include Area Code) (408)646-2095 — 22c OFFICE SYMBOL 52Mz

ORM 1473, 84 MAR — 83 APR edition may be used until exhausted; All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

☆ U.S. Government Printing Office 1986—606-243

RULE-BASED MOTION COORDINATION FOR THE ADAPTIVE SUSPENSION VEHICLE

S. H. Kwak and R. B. McGhee

Naval Postgraduate School
Department of Computer Science (Code 52)
Monterey, CA 93943, U.S.A.

ABSTRACT

This study investigates the utility of rule-based coordination of motion for rough-terrain locomotion by a hexapod walking machine. The logic for generating leg commands is written in Prolog while the simulation of the terrain and of the vehicle kinematics, as well as low level on-board computer functions, are written in extended Common Lisp. It is found that this approach results in code that is much easier to understand and modify than previous motion coordination programs written in Pascal. The authors believe that both the methodology and the stepping logic presented in this report possess sufficient merit to justify full-scale physical testing in the Adaptive Suspension Vehicle operated under DARPA contract by Ohio State University.

1. Introduction

The Adaptive Suspension Vehicle (ASV) is a large six-legged vehicle designed for outdoor operation in rough terrain. Limb motion coordination for the ASV is accomplished by an on-board computer network involving fifteen standard single-board computers as well as two special purpose computers [1, 2]. The software system is hierarchically organized with a clear distinction being made between an individual leg control level, a leg motion coordination level, and a body motion planning level [3]. Except for the two special purpose computers, the application software for the ASV is currently written almost entirely in Pascal. A custom designed real-time operating system, written mainly in PL/M, coordinates the functioning of all processes running on the various processors of the vehicle computer [4]. The total ASV software system involves somewhat more than one million bytes of code.

An important feature of the ASV is its omni-directional motion capability [1, 2] which gives it the general maneuverability characteristics of a helicopter. This behavior is achieved by providing the operator with a joystick with three major motion axes for control of vehicle forward velocity, lateral velocity, and turning velocity respectively [2]. The vehicle control computer accepts these commands and synthesizes a sequence of leg movements to produce the desired body behavior. It is assisted in this task by information from an optical terrain scanner which provides a map of terrain elevation in the immediate vicinity of the vehicle [5], and by force and position feedback from each leg.

Until now, nearly all outdoor experiments with the ASV have made use of a *tripod* gait in which legs are used in two sets of overlapping tripods [6, 7]. This gait was chosen both for its relative simplicity and because it is known to be uniquely optimal for high speed straight-line locomotion [7, 8, 9]. However, the tripod gait is not well suited to extreme terrain conditions in which a significant fraction of the area under a given leg of the ASV may be unsatisfactory for

load bearing due to the presence of rocks, holes, obstacles, soft soils, etc. In the latter case, simulation experiments [10, 11], and initial indoor testing [12], indicate that on-line optimization of leg sequencing should give better results.

Gaits involving real-time optimization of stability or maneuverability in the presence of terrain constraints are often called *free gaits* to distinguish them from the *periodic gaits* used by walking machines and animals in less difficult circumstances [13, 14]. Until now, all ASV experiments with free gaits have used an imperative language (Pascal) to encode stepping algorithms. However, because of the logical complexity of free gaits, the use of such a language produces code which is very difficult to understand or to change [15]. As a consequence, one of the authors adopted functional programming as implemented in Common Lisp for a simulation study of free gait algorithms [11]. While this was found to be helpful, and resulted in substantially more compact code, Common Lisp does not provide support for either object-oriented programming or logic programming [16]. Since optimization of stepping requires simulation of vehicle behavior, the use of object-oriented programming ought to produce more readable code. Fortunately, extended Common Lisp as implemented on Lisp machines possesses an object-oriented facility called *Flavors* [17]. As part of the work described in this report, the program documented in [11] was recoded using Flavors. The result, attached as an appendix, is believed by the authors to be much easier to comprehend and modify than the corresponding code of [11].

Although the use of Flavors represents a significant improvement in modularization and ease of understanding, the top level of leg motion planning is still difficult to comprehend. Further study has led the authors to the conclusion that this part of the ASV control problem fits a logic programming paradigm better than any other. That is, the logical conditions for transitions between various leg control states are best described by a set of *rules* [9, 18, 19]. This being the case, Prolog [20] was adopted for coding the top level of the coordination algorithm developed in this report. The resulting code is at least an order of magnitude shorter than the corresponding

Pascal code, and is remarkably easier to understand and modify.

The remainder of this report presents first a discussion of the mathematical model used in this study to simulate terrain and the ASV vehicle. This is followed by a description of the use of finite state machines for control of individual legs [18, 19]. Both these controllers and the vehicle and terrain models are simulated by the Lisp programs presented in the appendix. The next topic discussed is the use of Prolog to realize the free gait coordination algorithm. The report concludes with a discussion of the results of this investigation and suggestions for future research.

2. Vehicle and Terrain Model

While the vehicle model used in this study is based on the Adaptive Suspension Vehicle, it represents only the major vehicle dimensions and components. Specifically, the cabin and the terrain scanner are omitted from the simulation model, while the geometries of the body and the legs are identical to those of the ASV. Therefore, the simulation model is represented by a simple six-faced box with each leg drawn as two line segments as shown in Figure 1. The exact vehicle dimensional data can be found in other literature [2, 12].

The terrain adopted for this study consists of two types of cells. One type, called a *permitted* cell, is able to support the body load when a leg steps on it. The other type, named a *forbidden* cell, is not usable because of unfavorable terrain conditions. A typical terrain example utilized in this study is shown in Figure 1. A cell with an "x" mark is a forbidden cell while unmarked cells are permitted. As shown in Figure 1, the simulation terrain model is *prismatic* in nature. That is, the terrain height is determined by a function whose value is governed only by distance along a specified horizontal direction on the terrain. Forbidden cells on this terrain can be designated either manually by an operator or automatically by using a random number generator with a threshold chosen to produce a specified ratio between permitted cells and forbidden cells [11].

The dimensions of each cell are one foot by one foot when projected onto a horizontal plane. This size is comparable to that of the feet of the ASV, and is larger than the resolution of the terrain scanner [5, 12].

An overall block diagram of the program developed in this study is shown in Figure 2. This entire program is executed on a Symbolics 3650 Lisp machine [11, 21]. Each box shows an object that is an instance of a Flavor [17] with the exception of the Free Gait Coordinator which is written in Symbolics Prolog [21]. Like the physical ASV which has nine major parts, namely, body, vision sensor, cab, and six legs, the simulation object, "ASV" has correspondingly nine component objects, "Body", "Vision Sensor", "Joystick", and "Leg1" through "Leg6". These nine objects are linked to "ASV" through a *part* relation [22]. Each part has its sub-parts, and again links to them with a part relation. Differing from the nine major parts which have visible corresponding parts in the real ASV, the subparts of the simulation are not physically tangible, but are introduced because of their functionality for program development. For example, the "Leg1" object, which is a part of the "ASV" object, has six subparts : "Leg1 Plan Machine", "Leg1 Control Machine", "Leg1 Executor", "Leg1 Contact Sensor", "Leg1 Foothold Finder", and "Leg1 TKM Calculator". The "Leg1" object binds all of these subparts into one group with the part relation. In order to show the above relations among the objects in Figure 2, the six subpart objects are drawn under the "Leg1" object.

Besides the part relation, Figure 2 also shows the hierarchical control structure linking the simulation objects. Specifically, communication is restricted between objects in two adjacent levels by an assumption that upper levels have the right to access status information at lower levels, but the latter must receive explicit commands from upper levels to update their internal state. For example, when "ASV", the vehicle object, needs "Leg1" to support its body, it sends a "Place" decision to "Leg1" and continuously monitors "Leg1" as to whether "Leg1" has started to support the body or is in motion to try to reach a foothold. On receiving a "Place" decision

from "ASV", "Leg1" sends the "Place" decision to "Leg1 Plan Machine" while making observations of this machine. This type of message passing to and status observation from subordinates continues until the "Place" decision is accomplished. That is, when the foot of "Leg1" actually hits the ground, the contact sensor of "Leg1" detects the event and changes its internal state. The state change of "Leg1 Contact Sensor" is observed by "Leg1 Executor" and by "Leg1 Control Machine". In this way, the state change in the lowest level is propagated to higher levels until the touch down event arrives at "ASV ".

The joystick object simulates the physical three-axis joystick of the ASV through the use of six keys on the simulation computer keyboard to increment or decrement each of the three rates controlled by the joystick. These rates are *forward velocity*, *lateral velocity*, and *turn rate*, all in body coordinates. The altitude of the vehicle above the terrain and its orientation in roll and pitch relative to the terrain are automatically regulated using the algorithms described in [23].

While an elementary representation of the vision sensor is included in the appendix, it is not used in this study. Rather, as described in the above discussion of terrain, it is assumed that all forbidden cells have already been identified by prior terrain analysis. Of course this assumption does not represent a physical limitation of the ASV, but as made merely to allow this simulation to be focused on control of stepping, rather than on vision.

In addition to simplification of vision, this simulation also ignores leg mass in order to avoid the complexity of computing a center of gravity which moves with respect to the body. Moreover, all inertial forces are omitted from the simulation. That is, as in most previous simulation studies relating to gaits and control of stepping [8, 9, 10, 11, 24, 25, 26], only *static* stability is considered in this study. While this simplification would be serious in high speed locomotion, free gaits are most appropriate to low speed traversal of extremely difficult terrain, so the authors do not feel that this is a serious limitation on the applicability of the results of this investigation.

3. Finite State Control of Individual Legs

The basic approach to leg control used in this research was first proposed by Tomovic and McGhee [27], and involves the subdivision of leg motion into a number of discrete states [9, 18, 19]. In order to describe the application of this concept to the ASV, a number of definitions are needed as follows:

Definition 1: A *foothold* is a point on a piece of terrain, and can be assigned to a leg while the leg is in the air. When the foot of a leg is placed on the terrain, its assigned foothold becomes the *support point* of the leg. A foothold associated with a leg can be changed to a new one before the foothold becomes a support point [10].

Definition 2: The *support pattern* associated with a given set of leg support points is the convex hull of any point set in a horizontal plane which contains the vertical projections of all support points [13].

Definition 3: The magnitude of the *stability margin* at time t for an arbitrary support pattern is equal to the shortest distance from the vertical projection of the vehicle center of gravity to any point on the boundary of the support pattern. If the pattern is statically stable, the stability margin is positive. Otherwise, it is not defined [15].

Definition 4: A *working volume* is associated with each leg. This volume is a subset of three-dimensional space defined relative to the body and consists of the collection of points which can be reached by the foot of the given leg [11, 24].

Definition 5: A *temporal kinematic margin* is associated with each foothold. At any instant, this margin is the time remaining until the associated leg would reach the boundary of its working volume if the foothold were used as a support point [15, 24].

Evidently, supporting legs have their support points, and these points define a support pattern. The static stability of a walking machine is determined by the location of the center of gravity of its body with respect to the support pattern. The walking machine can change its body position only while the temporal kinematic margins of all supporting legs are larger than zero and the stability margin of the walking machine is positive. Therefore, the mobility of the walking machine is limited when either its stability is small or the temporal kinematic margin of any leg is small. This problem can be corrected by changing the support pattern or by changing the center of gravity of the machine; i.e., by leg placing and lifting or by body movement. The work presented in this report adopts both correction methods to enhance the mobility of a walking machine.

Because of the unevenness and obstacles associated with rough terrain, in this study leg movements during placing and lifting are restricted to be parallel to the direction of gravity in order to eliminate the possibility of striking obstacles with the side of the leg. From this consideration, the trajectory shown in Figure 3 is used for the free gaits discussed in this report. The seven segments of this trajectory define the states of the "Leg Control Machine" shown in Figure 4. Among these seven states, three states, "Ready", "Descent", and "Support" are asynchronous while the remaining four states, "Advance", "Contact", "Lift", and "Return" are synchronous. Among the three asynchronous states, two states, "Ready" and "Support", are introduced to handle the variable timing of lifting and placing events necessarily associated with free gaits. These two states are terminated by explicit commands from a higher level state machine, called the "Leg Plan Machine", shown in Figure 5. On the other hand, the "Descent" state of Figure 4 is designated as an asynchronous state for the reason that the time required for leg contact with the ground can be expected to deviate slightly from a nominal value, assumed to be 0.4 seconds, because of inaccuracies in measurement or mechanical errors in leg movement.

In the other four states, "Advance", "Contact", "Lift", and "Return", the accuracy of leg movement along the leg trajectory is not so critical as it is in the "Descent" state, so slight

position errors are acceptable. Thus, these four states are realized as synchronous states. Moreover, leg movement control based on only a timing event simplifies the control scheme. The quantities T1, T2, T3, and T4 shown in Figure 4 are the time duration of the four synchronous states, and represent reasonable amounts of time for the physical ASV leg to finish the associated movements.

As shown in Figures 4 and 5, the plan machines can send either a "Deploy_command" or a "Recover_command" to their subordinate leg control machines and monitor state changes of the latter. On the other hand, as shown in Figure 2, the plan machines are controlled and monitored by the leg objects. Thus, the role of the plan machines in the control hierarchy is to buffer the decisions of the free gait coordinator to allow for delays between leg motion planning and leg motion execution. Specifically, there is a time difference of one second between the leg motion planning done by the coordinator and the leg motion execution done by the leg executor under the control of the leg control machine. This is necessary because restoration of vehicle stability by placing a leg on the ground cannot be accomplished until a designated leg has moved to a designated foothold and physically begins to support the body. This prediction time represents the nominal time needed by a leg for such an action. Though leg sequencing could be planned a longer time ahead, this would require more computation and also would be subject to greater inaccuracy because of errors in predicting commands from the operator. Specifically, in this study, predicted body motion is derived from an assumption that the desired input commanded by an operator will not change within the next second. To make this assumption more reasonable, operator commands are filtered with a low pass filter before they are used by the program. This action also represents a simple approximation to the physical effect of ASV body inertia, since it prevents step changes in vehicle velocity.

Because the "Leg Plan Machine" is a Moore machine, as shown in Figure 5, labels on arcs from state to state are transition conditions, while arrows not terminating on states represent

outputs. The two outputs, "Deploy_command" and "Recover_command", are generated from the "Planned contact" and "Actual lift" states respectively, and are given to the lower level control machine. After sending out the "Deploy_command", the "Leg Plan Machine" continuously monitors the state change of its subordinate control machine. When the leg physically touches the ground, the control machine makes a state transition to the "Contact" state. The leg plan machine detects the control state transition, and makes its state transition to the "Eligible to lift" state. If the "Descent" control state lasts 0.4 seconds as planned by the control machine, then since the synchronous state "Advance" lasts 0.6 seconds, the "Planned contact" state of the plan machine lasts one second. Thus, the one second projection time is spent while the plan machine waits in the "Planned contact" state, and the plan machine is thereby synchronized with the physical touch-down event.

The "Eligible to lift" state means that the leg associated with the plan machine is ready to be lifted from the ground. The plan state transition to the "Eligible to lift" state is monitored through the leg object, and the information is collected by "ASV" so that it can provide the coordinator with the information that the leg is liftable from the ground. Thus, such a leg can be removed from the projected support pattern whenever the coordinator decides to do. Such a removal can occur in two different ways. One way is simple leg lifting. Whenever a leg is found to be redundant to making the vehicle stable, the coordinator can cause that leg to be lifted from the ground. The other possibility is that when the coordinator removes a given leg, it adds another leg into the projected support pattern. The latter case is called "leg exchange", and is intended to improve the mobility of the vehicle at the cost of a small increase in the complexity of the plan machines. Specifically, this action is useful when three legs are in a support pattern, but one leg is almost at its kinematic limit. In such a situation, the leg near its kinematic limit cannot be simply deleted from the support pattern because the vehicle would be unstable. On the other hand, without lifting the leg from the ground, the vehicle would soon come to stop because

of the leg reaching its kinematic limit. One way to solve this problem is to use two legs instead of one leg. That is, to cause one leg selected from the legs available to the coordinator to support the body, and at the same time to cause the other leg near its kinematic limit to be lifted from the ground. These two actions are performed simultaneously inside the coordinator, but two distinct decisions are sent to corresponding plan machines. First, the "Exchange_decision" is sent to the plan machine of the leg to be lifted so that the plan state is changed to "Planned exchange". Second, the "Place_decision" is given to the plan machine of the other leg to be placed so that the plan state is changed to "Planned contact". This action causes generation of a "Deploy_command" which is sent to the control machine so that the physical leg starts to move and eventually supports the body. The former leg plan machine will wait until it receives from its leg object the "Interlock_confirm" signal that is generated by the "ASV" when the physical leg moving toward the designated supporting point actually touches down on the ground. After arrival of the confirm signal, the plan machine changes its state to the "Actual lift" state, generates a "Recover_command", and sends the command to the lower level control machine. On receiving the command, the control machine changes its state to the "Lift" state, and passes its state to the lower level leg executor causing it to perform a physical leg lifting action.

Besides the "Interlock_confirm" signal, the "Support_state" signal from the control machine of the leg is also tested before the above state transition of the plan machine is made since a leg can be lifted from the ground only when its control state is the "Support" state. The last test in the plan state transitions is "Stable_without" which examines vehicle stability resulting from the leg lifting action. Since it is performed one second in advance, this test eliminates the possibility that the vehicle could become momentarily unstable because of lifting a leg.

Referring to Figure 5, the "Actual lift" state of the plan machine can be entered either from the "Planned exchange" state or from the "Planned lift" state. The first case is discussed above. For the latter state transition, two conditions are tested. One is whether the leg planned to be

lifted from the ground is in the "Support" control state, and the other is whether the leg can be lifted without making the vehicle unstable. The first test ensures a correct control state transition to the "Lift" control state. The second test takes care of the time difference between the leg lifting decision from the coordinator and the physical leg lifting action.

4. Leg Coordination Logic

The free gait coordinator located at the top of the control hierarchy continuously monitors the internal status of the robot object "ASV", and sends to it various commands to control the body and legs. In carrying out this action, the coordinator uses a free gait strategy which tends to maximize the number of legs in the air so long as the robot object is stable. This is done to increase the likelihood that the coordinator can find new legs to support its body when the robot mobility is limited either because of a small stability margin of its body, or because of a small temporal kinematic margin of one of the supporting legs. Thus, the role of the coordinator is similar to that of the brain of a horse carrying a human on its back when it walks over rough terrain. Like a horse that resists or modifies commands from a human operator under difficult conditions, the coordinator makes the robot object resist or modify operator commands depending on leg states, body attitude, and terrain conditions. Specifically, when one of the vehicle supporting legs has a small temporal kinematic margin, the coordinator resists the vehicle speed command from a human operator by reducing the vehicle speed in order to provide more time for leg recovery. After the leg with a small temporal kinematic margin is lifted from the ground, the coordinator then accelerates the vehicle until its speed reaches the desired value. The coordinator also modifies directional commands by perturbing the vehicle trajectory in a lateral direction to try to provide a larger stability margin when necessary.

The free gait coordinator is written in Symbolics Prolog and is listed in Figure 6. This pro-

gram is composed of three functional groups of predicates. The first group controls the flow of the whole program, while the second generates commands for the vehicle body and legs. The last group is responsible for bridging between the program written in Prolog and the program in Flavors. This is accomplished through the LISP function call facility provided by Symbolics Prolog. Specifically, anything following the "is" predicate in a Prolog clause may be either a Prolog arithmetic function or the name of a LISP function [21]. If a LISP function name follows the "is" predicate, it is evaluated according to its definition inside the LISP environment. In the program of Figure 6, all the names following the "is" predicates are names of LISP functions, and make connections to the LISP environment. A returned value resulting from a LISP function call may be used to instantiate a variable preceding the "is" Prolog predicate or to test whether the returned value matches a value preceding the "is" predicate. In the former case, the subgoal "is" always succeeds, but in the latter case, only when two values agree does the "is" subgoal succeed. For example, in Figure 6, the second clause, "initialize", has an "is" subgoal in its right side expression. The "is" predicate is followed by a LISP function "inits", and the "inits" function is evaluated inside the LISP environment. A returned value from the result of the function call is used to instantiate the dummy variable "X" without further usage, and the subgoal "initialize" is simply realized when the "inits" LISP function internally initiates a series of processes. Specifically, when the "inits" function is called, all the internal component objects of the robot object, "ASV", are instantiated from the Flavor definitions to complete the robot object. In this process, an empty "ASV", which exists before the "inits" LISP function is called, starts to fill its component slots with the body and six leg objects first, and then the body and the six legs fill their empty slots with sub-objects. This process continues until an object is reached which does not lack any sub-object necessary to fill its internal slots, such as the "Contact-Sensor" object in Figure 2. Thus, the "initialize" clause makes the robot object, "ASV", functionally complete to be used by the free gait coordinator.

The top level predicate of Figure 6, "robot", belongs to the first group of clauses, and provides the overall program control flow; i.e., it initializes the program and forms a loop for continuous program execution when the "robot" goal is typed from a terminal. The program loop is formed both by the built-in predicate "repeat" which provides a way to generate multiple solutions through backtracking, that is, by making the goal "repeat" always succeed on backtracking, and by the built-in predicate, "fail", which initiates backtracking whenever it is tested. Therefore, the "loop" subgoal is executed again and again. The "loop" subgoal is responsible for another level of control flow of the program so that the Prolog program repeats a series of operations, "get_command", "plan", and "execute"; i.e., it gets a desired velocity command from a human operator, plans vehicle motion based on the input command, and executes the planned vehicle motion. The "get_command" clause and the "execute" clause directly call their corresponding LISP functions, but the "plan" clause is refined into several subordinate Prolog clauses including "leg_plan" and "body_plan". The clauses or rules related to the plan predicate eventually communicate with the LISP environment to input states of the robot object, "ASV", and to output results from the planning process of the coordinator. Therefore, the data stored in the LISP robot object, "ASV", influences execution of the Prolog program, while the Prolog program modifys the data inside "ASV" using results from its execution. Thus, the whole plan portion of the free gait coordinator acts like a rule-based system being composed of three parts: inference engine, rules, and fact base. The plan clauses in Prolog and the status of the robot object, "ASV", in LISP, respectively, correspond to the rules and the fact base. The Prolog default search strategy corresponds to the inference engine, which searches its rules using a depth-first method until it finds a proper rule to fire. Therefore, in the Prolog program, the positions of Prolog clauses are interpreted as a priority preference among them. For example, the "leg_plan" portion consists of five "leg_plan" rules, and these rules are linearly ordered to express a free gait strategy that attempts to maximize the number of legs in the air. Thus, among the five rules,

the "lift_a_leg" rule is written first. The second "leg_plan" rule contains the second most favorable way to use legs. When this rule succeeds, the number of legs in the air is not changed, but the vehicle improves its mobility by exchanging a supporting leg that limits its movement for a leg in the air. The next most favorable method is to do nothing as long as the vehicle maintains its stability. This idea is written in the third "leg_plan" rule. The fourth way for "leg_plan" to succeed is to place a leg on the ground to maintain stability, although the number of legs in the air is reduced by this action. The last "leg_plan" rule represents the least favorable action since it both decreases the number of legs in the air and slows down the vehicle.

The top level "plan" clause makes the robot object, "ASV", follow a sequence of operations. First, it orders "ASV" to update its internal states. On receiving the "update_robot_state" command, "ASV" calculates the next body position based on the operator commands, and checks and collects internal state information for the coordinator, such as which legs are available to the coordinator. After completion of the updating process, the coordinator checks whether any leg is near its kinematic limit. If such a leg is found, the coordinator simply removes it from its support pattern, and designates the leg as a limit leg so that the "leg_plan" rules can provide a proper action to correct the leg limiting situation. In normal circumstances, the above mentioned leg is seldom found because, before a leg approaches its kinematic limit, the leg is lifted from the ground by the "leg_plan" rules that always try to minimize the number of supporting legs. However, sometimes this process is necessary when the vehicle directional command is abruptly changed or when the vehicle speed is too great. In the latter case, the legs on the ground quickly approach their temporal kinematic limit before the legs lifted from the ground are available to the coordinator. Therefore, the limit leg may be found either because of a planning error caused by an abrupt command change or because of temporal problems of lifted legs. After executing the "check_tkm_limit" clauses, the coordinator tests the first "leg_plan" rule, "lift_a_leg". In the "lift_a_leg" rule, the coordinator checks the vehicle stability first. If the vehicle is stable, it

selects the leg with the smallest temporal kinematic margin among the supporting legs because such a leg will be the next to limit the vehicle mobility. Thus, the "lift_a_leg" rule asks "ASV" whether there exists a leg with a smallest temporal kinematic margin by calling the LISP function, "smallest_tkm_leg", and obtains the leg information through a returned value from the LISP function call. The leg information obtained is stored in the local variable, "A_leg", and is used to determine if "A_leg" can be removed from the support pattern without causing static instability. If the vehicle is still stable without the leg, the coordinator causes this leg to be lifted from the ground by asserting a "lift" decision that will be used later by the "generate_decision" predicate to send out the decision to the "ASV". Therefore, when this rule succeeds, the number of legs in the air is increased. The second most favorable rule is "exchange_legs" which improves the mobility of the vehicle without affecting the number of legs in the air. This rule trys to exchange a leg, "LegA", which has the smallest temporal kinematic margin among the supporting legs for a leg, "LegB", which will provide the largest stability margin when the leg is added to the set of supporting legs. In this case, the stability margin associated with a leg is defined as the stability margin which results when the leg is placed on the terrain without changing legs in the support pattern except for excluding "LegA". Because "LegB" is selected based on the stability criterion, if it is placed, then the stability of the vehicle tends to be maximized by this action. However, this maximization is somewhat limited because at most three legs can be compared to find "LegB", and the designated foothold for each of these legs has been selected from footholds inside its working volume using the criterion of temporal kinematic margin. If "LegB" is found, then the "exchange_legs" rule does another test to determine whether the temporal kinematic margin of "LegB" is larger than that of "LegA" because, if "LegB" has a smaller temporal kinematic margin than that of "LegA", replacing "LegA" with "LegB" would cause the kinematic problem to become more serious rather than improved. If the above test succeeds, then a leg exchange is performed. Consequently, the mobility of the vehicle is improved without changing

the number of legs in the air.

The third most favorable "leg_plan" is doing nothing as long as the vehicle is stable, because without changing the number of legs in the air the vehicle can still move while following the desired vehicle commands. However, when this rule succeeds, the vehicle mobility is not improved. The fourth rule can succeed only if the stability test of the preceding rule fails. Since the planner runs one second in advance, this means that the vehicle is about to become unstable. In this case, the coordinator selects one leg from the available legs and causes it to support the body to restore stability. Though this action decreases the number of legs in the air, it is necessary to maintain the vehicle stability. When selecting such a leg, the coordinator tries to choose the leg which will give the largest stability margin among the available legs. Again, the vehicle stability is maximized in a limited sense. However, this attempt may or may not be successful, either because no leg available to the coordinator generates a sufficient stability margin if it were to be used to support the body, or because no legs are yet available to the coordinator. If a proper leg is found, it is commanded to support the body and the number of legs in the air is decreased. Therefore, this rule is least favorable to the coordinator among the above four "leg_plan" rules, but is necessary to maintain the vehicle stability.

The last "leg_plan" rule is "wait_for_legs", which both sacrifices the mobility of the vehicle and decreases the number of legs in the air. The first sub-rule is "try_new_foothold" which continuously assigns new footholds to the legs available to the coordinator until the coordinator finds a leg able to make the vehicle stable. This process tends to reduce the future mobility of the vehicle because the newly assigned foothold has a smaller temporal kinematic margin than that of the initially assigned foothold. To minimize the seriousness of this effect, new footholds are assigned in the order of their temporal kinematic margin. Therefore, when a foothold is found which can make the vehicle stable, it may have a larger temporal kinematic margin than the smallest temporal kinematic margin that the leg can provide, but it has a smaller temporal

kinematic margin than that provided by the initially assigned foothold.

The next sub-rule performs the "recovery" action. The central idea of this action is to help the "place_a_leg" and "try_new_foothold" rules so that they can find a suitable leg in the next control cycle. Because the leg placement rules handle one leg during each control cycle, sometimes the rules cannot find a suitable leg to solve a mobility problem of the vehicle, but by using two or three legs the problem could be solved. Instead of increasing the complexity of the "leg_plan" rules so that they can handle two or three legs in one control cycle, the problem is solved through multiple control cycles by introducing the "recovery" rule, while concurrently the speed of the vehicle is reduced in order to preserve its current stability. While simple, this approach is not guaranteed to find an ideal solution that could be obtained from generalized rules capable of handling multiple leg placements. When the "recovery" rule is tried, no single leg is able to make the vehicle stable using the previous rules. In this case, all the legs available to the coordinator are identically useless. Thus, any one leg can be arbitrarily chosen and made to support the vehicle body. This action causes the vehicle to have a new support pattern, and increases the chance that the coordinator can find an additional leg that makes the vehicle stable in the next control cycle. If such a leg is found in the next cycle, a two leg placement solution is found through two consecutive cycles the "leg_plan" rules. If an adequate leg is not found again, then the above process will be repeated once more with the result that all the available legs are consumed by the "recovery" rule, since at most three legs can be in the air at any given time.

The last sub-rule of "wait_for_legs" simply slows down the vehicle, and if a limit leg is found in the "check_tkm_limit" rules, then this rule restores the leg to the support pattern because the previous "leg_plan" rules could not make the vehicle stable without the limit leg. Moreover, the temporal kinematic margin of this leg will be increased because the vehicle speed is reduced by this rule. Specifically, if the limit leg information is asserted by the "check_tkm_limit" rules, this information is retracted from the Prolog data base by this rule so that the limit leg is not lifted

from the ground.

The next step in the "plan" rule is "body_plan" which is composed of two sub-goals. The first one is "speed_plan" which checks whether a vehicle speed reduction is requested from the "leg_plan" rules. If requested, it retracts the request and makes the vehicle slow down. If not, it makes the vehicle speed up to follow a desired speed command. The second sub-goal, "trajectory_plan", helps to increase the stability margin of the vehicle. Because the stability margin is omni- directional, there exists one shortest line segment between the center of the body and the boundary of the support pattern. The main idea of the "trajectory-plan" predicate is to increase the length of this line segment. The method adopted here is to push the center of the body away from the boundary of the supporting pattern in the direction parallel to the shortest line segment until the stability margin is larger than a specified desired value used for the stability test in the "leg_plan" rules. In order to implement the above idea, a "recovery vector" is defined such that it points away from the boundary of the supporting pattern along the line segment that determines the stability margin at that moment. The magnitude of the vector is proportional to the reciprocal of the stability margin. The recovery vector is internally interpreted as a recovery velocity, and is superimposed on the operator velocity command so that the vehicle is pushed away from the boundary. More precisely, only the component of the recovery vector perpendicular to the velocity command is used in order to eliminate a speed disturbance in the direction commanded by an operator during such a *push* operation.

After planning leg and body motions, the coordinator sends decisions to the robot object, "ASV", one by one, until all the decisions in the Prolog data base are exhausted. Specifically, two activities, a decision retraction from the Prolog data base and the decision dispatch to "ASV", are repeated by the "fail" predicate at the end of the "generate_decision" clause until all the decisions are popped from the Prolog data base. When all the decisions are thus removed from the data base, the second "generate_decision" clause is tested to determine whether or not a limit leg

exists in the data base. If this is true, then the second clause deletes the limit leg from the data base and sends a lift decision to the robot object, "ASV".

In the execution process, the coordinator instructs "ASV" to execute the decisions given to it in the planning process. The six leg plan machines, the six leg control machines, the six leg executors, and the six contact sensors start functioning according to the hierarchy shown in Figure 2. The decisions from the coordinator cause state transitions of the six leg plan machines, and the state machines send commands to their subordinate control machines so that the control machines can update their control states. The control machines instruct their leg executors to move their legs depending on the control states and monitor states of their contact sensors to check leg touch-down events. Finally, the coordinator calls the "graphical_display" LISP function to draw an updated robot image on the screen of the simulation computer. Thereby one control cycle is completed. This action continues until the operator interrupts the program.

5. Discussion

The gait selection problem for a walking machine is inherently an ill-defined problem [28] because neither the goal state, nor the rules (or operators) available for reducing the state difference to reach the goal state are clearly specified. Instead, two general minimality criteria are known: stability margin and temporal kinematic margin. Thus, it is necessary to add more structure to this problem using constraints in order to make it more manageable. A frequently used constraint is to make a walking machine use a periodic leg stepping sequence, follow a straight-line trajectory, and walk on flat level terrain free from prohibited stepping areas. Using this constraint, a family of well understood optimal gaits, called *wave gaits* is found [7, 9]. However, for more general applications of walking machines, such as locomotion on rough terrain along an arbitrary trajectory, this constraint is too restrictive. Rather, it should be either relaxed or eliminated. If the constraint is removed, free gaits result. Thus, free gaits include all possible periodic gaits. In the specific free gait implementation of this study, instead of limiting the

stepping sequence and body trajectory, an overall strategy is added to structure the gait problem; namely, to minimize the number of supporting legs to the greatest extent possible.

Differing from a tripod gait that exists as a closed-form solution, free gaits are generated by on-line optimization depending on the situation currently confronted by a walking machine. Therefore, the first performance measure of any free gait coordination algorithm is whether it reduces to a tripod gait in situations where this gait is known to be optimal. From the computer simulation, it is observed that the gait generated by the coordination algorithm when it drives the ASV along a straight line on a flat level terrain containing no obstacles is essentially a tripod gait. Moreover, when a small number of obstacles are added to the terrain, the program still uses tripod-like gaits. However, as the percentage of forbidden cells on the terrain is increased, as would be expected, the gaits generated by the program change.

Perhaps the most distinctive departure of leg stepping from tripod gaits is observed when the ASV makes a turn-in-place motion. Ideally, during this motion, the middle legs should show less frequent leg movements compared with those of the front and the rear legs. Evidently, tripod gaits cannot exhibit such leg motions since the middle legs are cycled at the same rate as the front and rear legs. The free gaits generated by the program clearly show the expected behavior. Specifically, during a turning-in-place motion of 90 degrees or more, the middle legs are not usually changed during several motion cycles of the front and back legs.

To obtain an objective performance evaluation, the model vehicle was made to walk over terrain containing randomly generated obstacles while following a standard trajectory. Along this trajectory, three types of motion are sequentially performed. First of all, after the vehicle is initialized, it moves in the forward direction. It then makes a 90 degree turn-in-place at the middle of the terrain in the counter-clockwise direction, and finally performs side stepping until it reaches the edge of the model terrain [11]. For these three trajectory segments, the vehicle is sequentially commanded with three different velocity inputs: a one foot per second velocity input

in the forward direction, a one-twentieth radian per second turning velocity input, and a half foot per second side-stepping velocity input in the right side direction. A typical terrain example is shown in Figure 1. In this example, eighty percent of the area is classified as permitted cells.

Performance tests were based on trials on 10 different terrains with the same probability of occurrence of permitted cells. The general tendency of the results obtained in this evaluation is as would be expected; namely, the larger the probability of permitted cells, the better the chance that a vehicle can cross the randomly generated terrain. No failures to complete the standard trajectory were observed for any terrain in which permitted cells occupied 60 percent or more of the area of the whole simulation terrain. For terrain with 50 percent, 40 percent, and 30 percent permitted cells, the tests showed 8, 9, and 8 successes out of ten trials respectively. However, the success rate was sharply reduced for terrain containing less than 20 percent permitted cells, and no successful motion was observed when 10 percent or less of the total terrain cells was permitted. Therefore, practically, when more than 30 percent of the total terrain cells are permitted, the program makes the ASV maneuver without great difficulty. The average time spent to run one test with a Symbolics 3675 Lisp Machine is about 15 minutes, almost 10 times slower than the simulation time on which all the internal calculations are based.

One of the advantages of rule-based control of stepping is the ease of modification of stepping logic resulting from the fact that each rule defines a small, relatively independent piece of behavior. To demonstrate this advantage, two alternative rule sets were compared to the original rule set. In the first experiment, the "exchange_legs" rule was removed from the "leg_plan" rules because this rule was suspected of being redundant. When this is done, the "lift_a_leg" rule and the "place_a_leg" rule implicitly share leg exchange responsibility. However, it was found that the performance of the new rule configuration is poorer than that of the complete rules. More frequent planning failure is observed on randomly generated complicated terrain. Gaits generated by the new rules set are not so close to tripod gaits as those generated by the complete

rules, although during a turn-in-place motion the new rule set also clearly treats the middle legs differently from the other four legs. Therefore, it can be concluded that the "exchange_legs" rule improves performance. This rule provides a shortcut of the logical process and works somewhat like compiled knowledge in a human or animal brain obtained from experience.

If the "lift_a_leg" rule is also removed from the "leg_plan" rules, the coordinator instructs the ASV to use as many legs as possible to support its body. Thus, the resulting leg stepping strategy has almost an opposite meaning compared to the original one. As would be expected, the gaits generated by the rules without these two leg plan rules have little resemblance to tripod gaits, even on flat level terrain. The overall performance of the simplest rule set on rough terrain is much better than that of the former simplified rule set, but slightly inferior to that of the original rules. It frequently slows down the vehicle, but the terrain adaptability to rough terrain is almost the same as that of the original rule set.

The overall conclusion from simulator tests to date is that the original rule set is a good one. Specifically, when the ASV is presented with with easy terrain and essentially forward motion, a tripod gait is produced by the free gait coordinator. On the other hand, for turning in place or moving over terrain with few available footholds, the free gait algorithm seems to make "intelligent" decisions and displays a remarkable ability to pick its way through regions with sparse footholds. Attempts to simplify the rule set presented have so far resulted in deterioration of their behavior.

6. Summary and Recommendations

The main purpose of this study was to demonstrate the value of a multiple paradigm programming environment in the development of software for coordination of motion for the ASV walking machine. An important secondary goal was using such a facility to better explain the algorithm developed in [11], and to investigate the effect of changes in its basic strategy. With respect to the first objective, the authors believe that the code contained in this report is much easier to read and to modify than the version presented in [11] and [15], which used, respectively, functional programming (Common Lisp), and imperative programming (Pascal). With regard to the second objective, we find Prolog to be more powerful than English in explaining the logical conditions for state transitions in the leg plan finite state machines used for implementing the coordination algorithm. It is especially important that the rules of Figure 6 can be read declaratively by humans and procedurally by a computer. One of the consequences of this fact is that modification of stepping algorithms is remarkably easier in Prolog than in Lisp or Pascal.

It is our belief that our simulation studies to date have shown that the full free gait algorithm developed in this report is appropriate for physical experiments with the ASV. We believe that the multilingual approach we have used should be retained in such experiments, and that future vehicle computer upgrades should provide a platform for efficient execution of Prolog and extended Common Lisp. Finally, we recommend that program development continue to be done on a Lisp machine with facilities for downloading to a runtime system in the vehicle computer.

Among studies remaining to be conducted are inclusion of vehicle inertia in the simulation, effects of leg motion on the location of the vehicle center of gravity, and a better simulation of the vision system. Also, in the experiments reported herein, a constant small value for the minimum vehicle stability and for the minimum temporal kinematic margin was used. It is important to determine how variation in these threshold values affects safety and maneuverabil-

ity, especially when vehicle inertia is taken into account. Such a study will be undertaken early in the next phase of this research along with an investigation of further changes to the Prolog rule set used for the top level of motion coordination.

References

[1] McGhee, R. B., "Computer Coordination of Motion for Omni-Directional Hexapod Walking Machines," *Advanced Robotics,* Vol. 2, No. 1, pp. 91-99, October 1986.

[2] Waldron, K. J. and McGhee, R. B., "The Adaptive Suspension Vehicle," *IEEE Controls Magazine,* Vol. 6, No. 6, pp. 7-12, December 1986.

[3] McGhee, R. B., Orin, D. E., Pugh, D. R., and Patterson, M. R., "A Hierarchically-Structured System for Computer Control of a Hexapod Walking Machine," *Theory and Practice of Robots and Manipulators,* pp. 375-381, ed. by A. Morecki et al, Hermes Publishing, 1985.

[4] Schwan, K., Bihari, T., Weide, B. W., and Taulbee, G., "High-Performance Operating System Primitives for Robots and Real-Time Control Systems," *ACM Transactions on Computer Systems,* Vol. 5, No. 3, August 1987, pp 189-231.

[5] Klein, C. A., Kau, C. C., Ribble, E. A., and Patterson, M. R., "Vision Processing and Foothold Selection for the ASV Walking Machine," *Proceedings of SPIE Conference - Advances in Intelligent Robotic Systems,* Cambridge, MA, November 1-6, 1987.

[6] Klein, C. A., Olson, K. W., and Pugh, D. R., "Use of Force and Attitude Sensors for Locomotion of a Legged Vehicle over Irregular Terrain," *International Journal of Robotics Research,* Vol. 2, No. 2, Summer 1983, pp. 3-17.

[7] Song, S. M. *Kinematic Optimal Design of a Six-Legged Walking Machine,* Ph. D. dissertation, The Ohio State University, Columbus, Ohio, 1984.

[8] Bessonov, A. P. and Umnov, N. V., "The Analysis of Gaits in Six-Legged Vehicles According to Their Static Stability," *Proceedings of CISM-IFTOMM Symposium on Theory and Practice of Robots and Manipulators,* Udine, Italy, September 1973.

[9] McGhee, R. B., "Robot Locomotion," in *Neural Control of Locomotion,* pp. 237-264, ed. by R. M. Herman, et al., Plenum Press, New York, 1976.

[10] McGhee, R. B. and Iswandhi, G. I., "Adaptive Locomotion of a Multilegged Robot over Rough Terrain," *IEEE Transactions on Systems, Man, and Cybernetics,* Vol. SMC-9, No. 4, pp. 176-182, April 1979.

[11] Kwak, S. H., *A Computer Simulation Study of a Free Gait Motion Coordination Algorithm for Rough-Terrain Locomotion by a Hexapod Walking Machine,* Ph. D. dissertation, The Ohio State University, Columbus, Ohio, 1986.

[12] Waldron, K. J., et al., *A Mobility System Design Study for an Agile Autonomous Land Vehicle,* Final Technical Report, Contract DAAE07-84-K-R001, The Ohio State University, Columbus, Ohio, May 1988.

[13] McGhee, R. B., "Walking Machines," in *Advances in Automation and Robotics,* pp. 259-284, ed. by G. N. Saridis, Jai Press, Inc. 1985.

[14] Pearson, K. G. and Franklin, R., "Characteristics of Leg Movements and Patterns of Coordination in Locusts Walking on Rough Terrain," *International Journal of Robotics Research,* Vol. 3, No. 2, Summer 1984.

[15] Kwak, S. H., *A Simulation Study for Free-Gait Algorithms for Omni-Directional Control of Hexapod Walking Machines,* M. S. thesis, The Ohio State University, Columbus, Ohio, 1984.

[16] Steele, G. L., *Common Lisp,* Digital Press, Maynard, Massachusetts, 1984.

[17] Bromley, H. and Lamson, R. *Lisp Lore,* 2nd edition, Kluwer Academic Publishers, New York, 1987.

[18] McGhee, R. B., "Finite State Control of Quadruped Locomotion," *Simulation,* Vol 9, No. 3, pp. 135-140, September 1967.

[19] Frank, A. A., *Automatic Control Systems for Legged Locomotion Machines,* Ph. D. dissertation, University of Southern California, Los Angeles, California, 1968.

[20] Clocksin, W. F., and Mellish, C. S., *Programming in Prolog,* 3rd edition, Springer-Verlag, New York, 1987.

[21] Anon., *User's Guide to Symbolics Prolog,* Symbolics, Inc., Concord, Massachusetts, September 1986.

[22] Winston, P. H., *Artificial Intelligence,* 2nd edition, Addison-Wesley, Reading, Massachusetts, 1984.

[23] Lee, W. J., and Orin, D. E., "The Kinematics of Motion Planning for Multilegged Vehicles over Uneven Terrain," *IEEE Transactions on Robotics and Automation,* Vol. RA-4, No. 3, April 1988.

[24] Lee, W. J., and Orin, D. E., "Omnidirectional Supervisory Control of a Multilegged Vehicle Using Periodic Gaits," *IEEE Transactions on Robotics and Automation,* Vol. RA-4, No. 2, August 1988.

[25] Hirose, S., Fukuda, Y., and Kikuchi, H., "The Gait Control System of a Quadruped Walking Machine," *Advanced Robotics,* Vol. 1, No. 4, December 1986, pp. 289-323.

[26] Klein, C. A. and Messuri, D. A., "Automatic Body Regulation for Maintaining Stability of a Legged Vehicle During Rough-Terrain Locomation," *IEEE Journal of Robotics and Automation,* Vol. RA-1, No. 3, pp. 132-141, September 1985.

[27] Tomovic, R. and McGhee, R. B., "A Finite State Approach to the Synthesis of Bioengineering Control Systems," *IEEE Transactions on Human Factors in Electronics,* Vol. HFE-7, No. 2, pp. 65-69, February 1966.

[28] Matlin, M., *Cognition,* Holt, Rinehart, and Winston, New York, 1983.

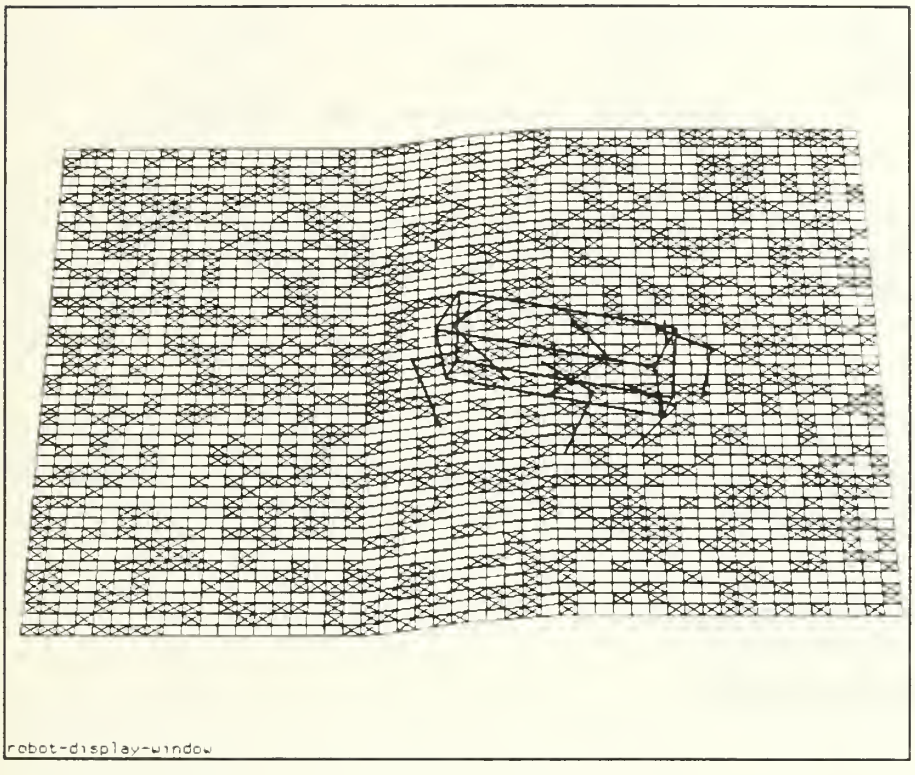


Figure 1: Typical Simulation Terrain and Vehicle

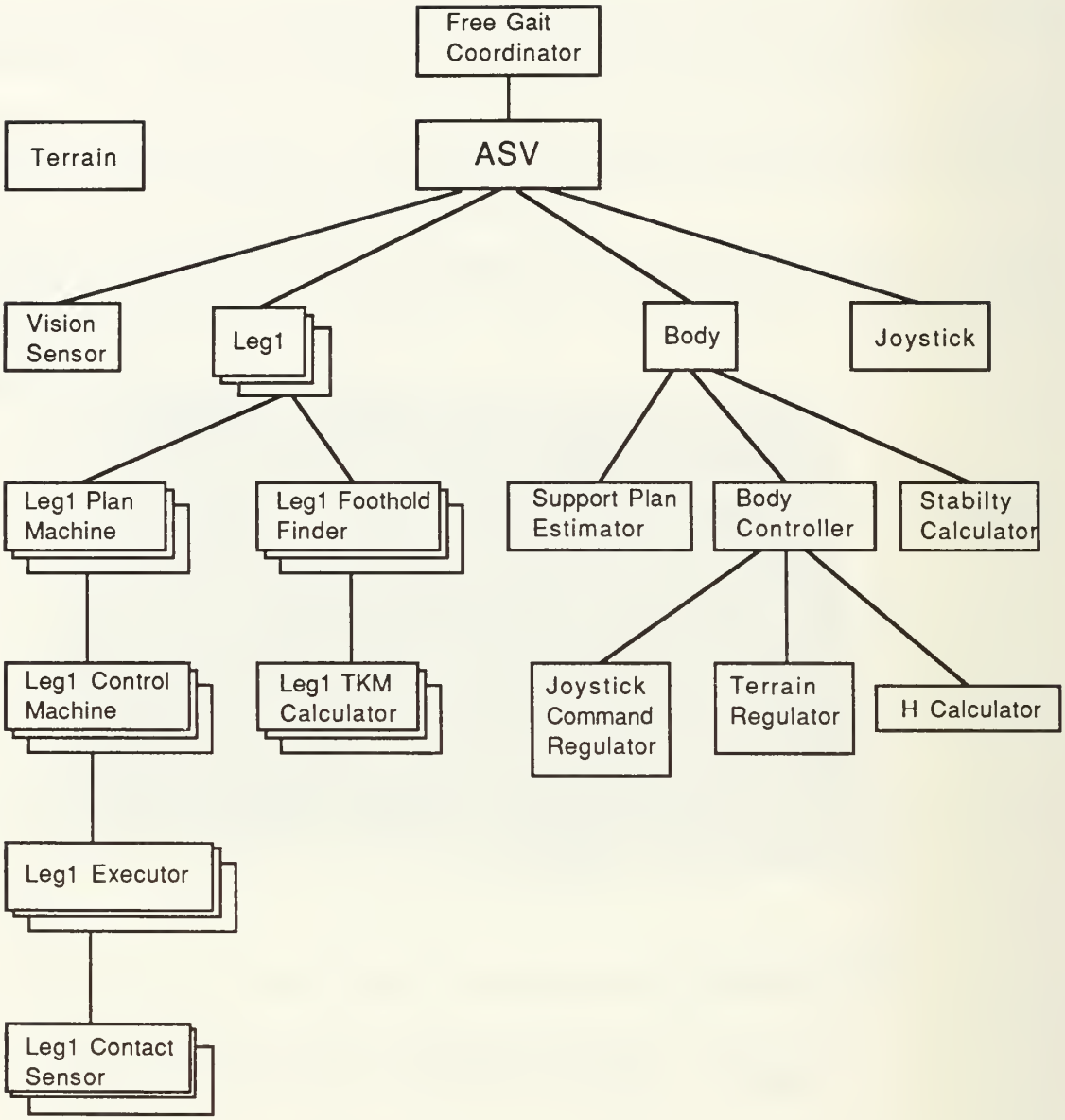


Figure 2: Hierarchy of simulation objects

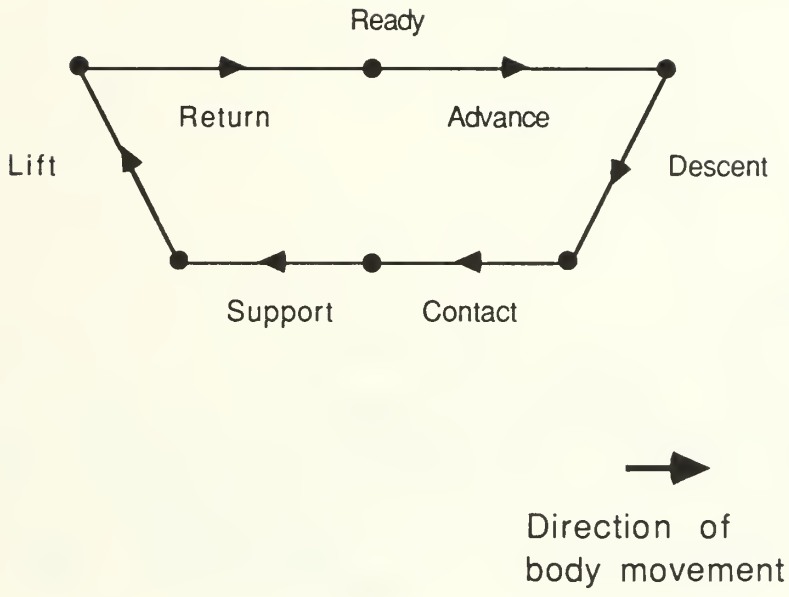


Figure 3. Leg motion relative to body during forward body motion over level terrain

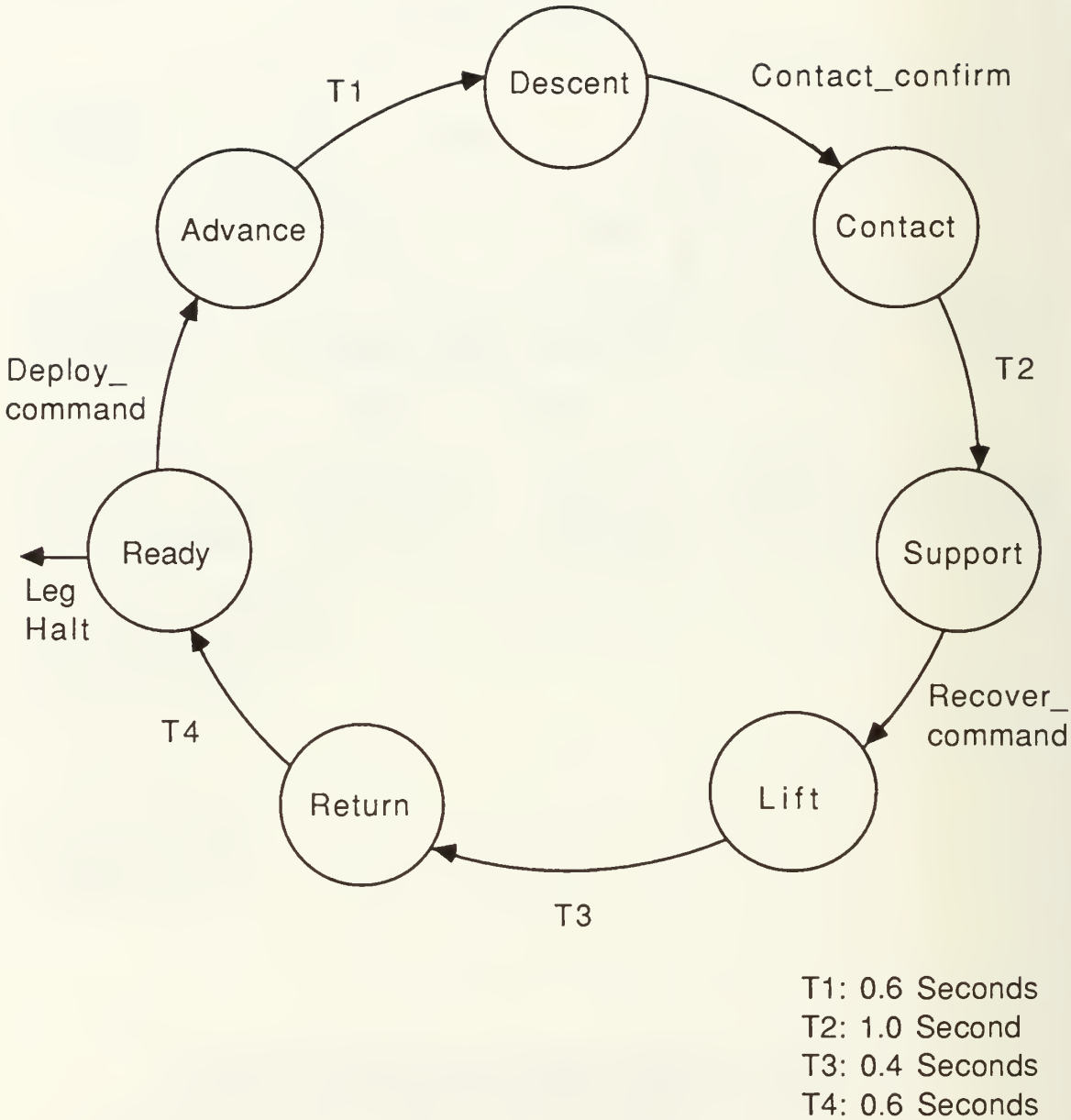


Figure 4: State diagram for Leg Control Machine

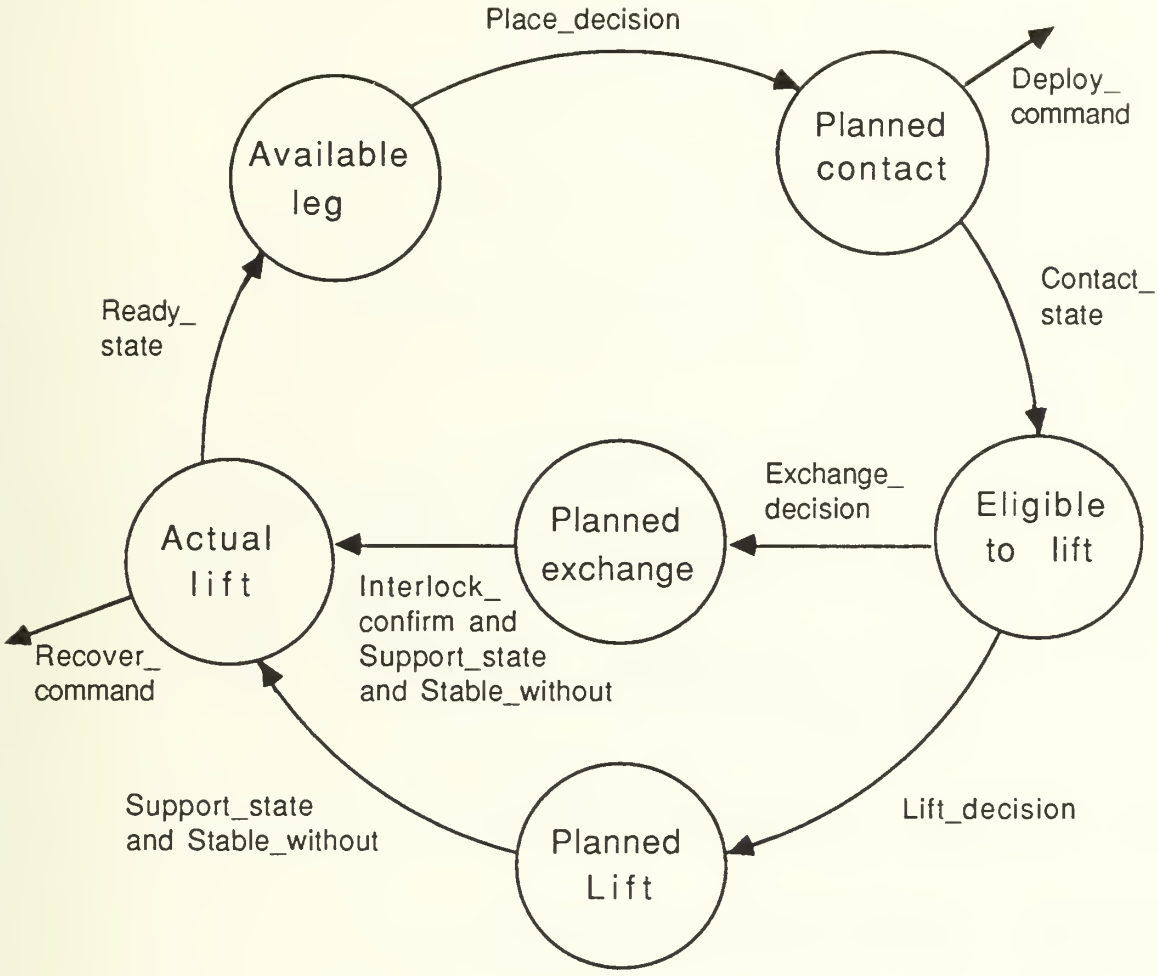


Figure 5: State diagram for Leg Plan Machine

```
%%%  -*- Mode: PROLOG; Package: robot-rules; Default-character-style: (:FIX :ROMAN :L
RGE); -*-
robot :- initialize, repeat, loop, fail.

initialize :- X is inits.

loop :- get_command, plan, execute, !.

get_command :- X is read_joystick.

plan :- update_robot_state, check_tkm_limit,
        leg_plan, body_plan, generate_decision, !.

update_robot_state :- X is update_robot_status.

check_tkm_limit :- A_leg is at_tkm_limit, A_leg \== nil,
                   asserta(limit_leg(A_leg,lift)).
check_tkm_limit.

leg_plan :- lift_a_leg.
leg_plan :- exchange_legs.
leg_plan :- stable.
leg_plan :- place_a_leg.
leg_plan :- wait_for_legs.

stable :- Condition is stable_p, Condition == t.

lift_a_leg :- stable, A_leg is smallest_tkm_leg, A_leg \== nil,
              Condition is stable_without(A_leg), Condition == t,
              asserta(decision(A_leg,_,lift)).

exchange_legs :- stable, LegA is smallest_tkm_leg, LegA \== nil,
                 LegB is max_sm_leg(LegA), LegB \== nil,
                 Condition is has_more_tkm(LegB,LegA),
                 Condition == t,
                 asserta(decision(LegA,LegB,exchange)).

place_a_leg :- A_leg is max_sm_leg(_), A_leg \== nil,
               asserta(decision(A_leg,_,place)).

wait_for_legs :- try_new_foothold.
wait_for_legs :- recovery, asserta(reduce_speed).
wait_for_legs :- asserta(reduce_speed), restore_limit_leg.

try_new_foothold :- A_leg is leg_with_new_foothold, A_leg \== nil,
                    asserta(decision(A_leg,_,place)).

recovery :- A_leg is do_recovery, A_leg \== nil,
            asserta(decision(A_leg,_,place)), restore_limit_leg.

restore_limit_leg :- retract(limit_leg(A_leg,lift)).
restore_limit_leg.
```

Figure 6: Free Gait Coordinator

```
body_plan :- speed_plan, trajectory_plan.

speed_plan :- retract(reduce_speed), slow_down.
speed_plan :- speed_up.

speed_up :- X is speed_up_robot.

slow_down :- X is slow_down_robot.

trajectory_plan :- stable_m, restore_trajectory.
trajectory_plan :- modify_trajectory.

stable_m :- Condition is stable_p_m, Condition == t.

restore_trajectory :- X is restore_command.

modify_trajectory :- X is modify_command.

generate_decision :- retract(decision(A_leg,B_leg,A_decision)),
                     X is send_decision(A_leg,B_leg,A_decision), fail.
generate_decision :- retract(limit_leg(A_leg,A_decision)),
                     X is send_decision(A_leg,_,A_decision), fail.
generate_decision.


execute :- execute_motion, draw_robot, !.

execute_motion :- X is execute_planned_motion.

draw_robot :- X is graphical_display.
```

Figure 6: continued...

Appendix

Lisp Code for ASV Simulation

```
; -*- Mode: LISP; Package: BODY; Syntax: Common-lisp -*-

****************************************************************

 body-controller definition

****************************************************************

efflavor body-controller(joystick-command-regulator terrain-regulator
                         H-calculator
                         body-trans-rate1 body-rotate-rate1
                         body-trans-rate6 body-rotate-rate6
                         body-trans-rate10 body-rotate-rate10
                         H1 inv-H1 H6 inv-H6 H10 inv-H10
                         H inv-H body-t body-r)

         ()
:initable-instance-variables)



efmethod (init body-controller)
         ()
(setf joystick-command-regulator (make-instance 'joystick-command-regulator))
(setf terrain-regulator (make-instance 'terrain-regulator))
(setf H-calculator (make-instance 'H-calculator))
(init joystick-command-regulator)
(init terrain-regulator)
(setf H (init H-calculator))
(init-body-rates self)
(init-H self)
H1
)


defmethod (init-body-rates body-controller)
         ()
(setf body-trans-rate1 '(0 0 0))
(setf body-trans-rate6 '(0 0 0))
(setf body-trans-rate10 '(0 0 0))
(setf body-rotate-rate1 '(0 0 0))
(setf body-rotate-rate6 '(0 0 0))
(setf body-rotate-rate10 '(0 0 0))
)



defmethod (init-H body-controller)
         ()
library fucntion : ident
(setf H1  H)
(setf H6  H)
(setf H10 H)
(setf inv-H (matrixinv H))
(setf inv-H1 inv-H)
(setf inv-H6 inv-H)
(setf inv-H10 inv-H))
```

```
(defmethod (control body-controller)
           (joystick-command deceleration-factor estimated-support-plane)
  (setf H H1)
  (update self joystick-command deceleration-factor estimated-support-plane)
  (save self)
  (dotimes (i 10)
    (cond ((equal i 0)
           (setf body-trans-rate1 body-t)
           (setf body-rotate-rate1 body-r)
           (setf H1 H)
           (setf inv-H1 inv-H))
          ((equal i 5)
           (setf body-trans-rate6 body-t)
           (setf body-rotate-rate6 body-r)
           (setf H6 H)
           (setf inv-H6 inv-H))
          ((equal i 9)
           (setf body-trans-rate10 body-t)
           (setf body-rotate-rate10 body-r)
           (setf H10 H)
           (setf inv-H10 inv-H)))
    (update self joystick-command deceleration-factor estimated-support-plane)
    )
  (restore self))



(defmethod (update body-controller)
           (joystick-command deceleration-factor estimated-support-plane)
; internally used by control method
  (let* ((t-command (regulate terrain-regulator
                              estimated-support-plane H))
         (j-command (regulate joystick-command-regulator
                              joystick-command deceleration-factor))
         )
    (setf body-t (list (first j-command) (second j-command)
                       (third t-command)))
    (setf body-r (list (first t-command) (second t-command)
                       (third j-command)))
    (setf H (new-H H-calculator body-t body-r))
    (setf inv-H (matrixinv H))))



(defmethod (restore body-controller)
           ()
; internally used by control method
  (restore joystick-command-regulator)
  (restore terrain-regulator)
  (restore H-calculator))



(defmethod (save body-controller)
           ()
; internally used by control method
  (save joystick-command-regulator)
  (save terrain-regulator)
  (save H-calculator))
```

```
efmethod (get-body-trans-rate1 body-controller)
         ()
body-trans-rate1)


efmethod (get-body-rotate-rate1 body-controller)
         ()
body-rotate-rate1)


efmethod (get-body-trans-rate10 body-controller)
         ()
body-trans-rate10)


efmethod (get-body-rotate-rate10 body-controller)
         ()
body-rotate-rate10)


efmethod (get-H1 body-controller)
         ()
H1)


efmethod (get-inv-H1 body-controller)
         ()
inv-H1)


efmethod (get-H6 body-controller)
         ()
H6)


efmethod (get-inv-H6 body-controller)
         ()
inv-H6)


efmethod (get-H10 body-controller)
         ()
H10)


efmethod (get-inv-H10 body-controller)
         ()
inv-H10)
```

```
; -*- Mode: LISP; Syntax: Common-lisp; Package: BODY -*-

************************************************************

 body flavor definition

************************************************************



efflavor body(stability-calculator support-plane-estimator
                                    body-controller owner
                                    estimated-support-plane
                                    deceleration-factor
                                    support-plane-clock
                                    modify-vector
                                    modify-vector-p
                                    stop-motion-flag
                                    joy-command)
         ()
:initable-instance-variables)


efmethod (slow-down body)
         ()
(setf deceleration-factor (+ deceleration-factor 1))
(if (> deceleration-factor 20)
    (setf deceleration-factor 20)))

efmethod (speed-up body)
         ()
(setf deceleration-factor (- deceleration-factor 1))
(if (< deceleration-factor 0)
    (setf deceleration-factor 0)))



efmethod (stable-m body)
         (supporting-legs)
(stable-m stability-calculator
          supporting-legs (get-H10 body-controller)))


efmethod (stable-p-m body)
         (supporting-p-legs a-leg)
(stable-p-m stability-calculator
                    supporting-p-legs
                    (get-H1 body-controller)))


efmethod (stop-p body)
         ()
(let ((trans-rate (get-body-trans-rate1 self)))
  (equal (list (first trans-rate)
               (second trans-rate))
         '(0.0 0.0))))


efmethod (modify-command body)
         ()
(setf modify-vector
      (get-recovery-vector stability-calculator)))
```

```
(defmethod (modify-command-p body)
           ()
  (setf modify-vector-p
        (get-recovery-vector-p stability-calculator)))


(defmethod (restore-command body)
           ()
  (setf modify-vector '(0 0 0)))



(defmethod (restore-command-p body)
           ()
  (setf modify-vector-p '(0 0 0)))


(defmethod (stop-motion body)
           (a-leg)
  (setf stop-motion-flag a-leg))



(defmethod (restore-motion body)
           ()
  (setf stop-motion-flag nil))




(defmethod (init body)
           ()
  (setf deceleration-factor 0)
  (setf modify-vector-p '(0 0 0))
  (setf modify-vector '(0 0 0))
  (setf stop-motion-flag nil)
  (setf support-plane-clock 10)
  (setf stability-calculator
        (make-instance 'stability-calculator))
  (setf support-plane-estimator
        (make-instance 'support-plane-estimator))
  (setf body-controller
        (make-instance 'body-controller))
  (init stability-calculator)
  (init support-plane-estimator)
  (init body-controller)
  )

(defmethod (get-modify-vector body)
           ()
  (vectsub modify-vector
           (dotprod modify-vector
                    (normalize-vector joy-command))))



(defmethod (get-modify-vector-p body)
           ()
  modify-vector-p)
```

```
lefmethod (calculate-motion body)
          (joystick-command legs)
 (setf joy-command joystick-command)
 (cond ((equal support-plane-clock 10)
        (setf estimated-support-plane
              (get-plane support-plane-estimator legs))
        (setf support-plane-clock 0)))
 (setf support-plane-clock (+ support-plane-clock 1))
 (cond
   ((or stop-motion-flag (null modify-vector-p))
    (control body-controller
             '(0 0 0)
             0 estimated-support-plane))
   (modify-vector-p
    (control body-controller
             (vectadd joy-command (get-modify-vector-p self))
             deceleration-factor estimated-support-plane))
   (t
    (control body-controller
             (vectadd joy-command (get-modify-vector self))
             deceleration-factor estimated-support-plane))))


lefmethod (get-estimated-support-plane body)
          ()
 estimated-support-plane)


lefmethod (get-body-trans-rate1 body)
          ()
 (get-body-trans-rate1 body-controller))


lefmethod (get-body-rotate-rate1 body)
          ()
 (get-body-rotate-rate1 body-controller))


lefmethod (get-body-trans-rate10 body)
          ()
 (get-body-trans-rate10 body-controller))


lefmethod (get-body-rotate-rate10 body)
          ()
 (get-body-rotate-rate10  body-controller))


lefmethod (get-H1 body)
          ()
 (get-H1 body-controller))


lefmethod (get-inv-H1 body)
          ()
 (get-inv-H1 body-controller))
```

```
(defmethod (get-H6 body)
           ()
  (get-H6 body-controller))


(defmethod (get-inv-H6 body)
           ()
  (get-inv-H6 body-controller))


(defmethod (get-inv-H10 body)
           ()
  (get-inv-H10 body-controller))




(defmethod (more-stable body)
           (supporting-legs leg1 leg2)
  (more-stable stability-calculator
               supporting-legs (get-H10 body-controller)
               leg1 leg2))


(defmethod (stable body)
           (supporting-legs)
  (stable stability-calculator
          supporting-legs (get-H10 body-controller)))


(defmethod (stable-p body)
           (supporting-p-legs)
  (stable-p stability-calculator
          supporting-p-legs (get-H1 body-controller)))
```

```
; -*- Mode: LISP; Syntax: Common-lisp; Package: BODY -*-

;*********************************************************

; regulator flavor definition

;*********************************************************


(defflavor regulator((max-a 3.2174) (time-const 0.5) (sampling-time 0.1))
        ()
:initable-instance-variables)


(defmethod (filter regulator)
        (desired-x present-x)
first order regulation
(let ((del-vel (/ (- desired-x present-x) time-const)))
  (+ (* (g-limitor self del-vel) sampling-time)
     present-x)))


(defmethod (g-limitor regulator)
        (del-vel)
limit acceleration to 3.2174 ft/(sec*sec)  or 0.1 G.
(cond ((> del-vel max-a)   max-a)
      ((< del-vel (- max-a)) (- max-a))
      (T del-vel)))




;*********************************************************

; joystick-command-regulator flavor definition

;*********************************************************


(defflavor joystick-command-regulator(body-trans-rate-x
                                      body-trans-rate-y
                                      body-rotate-rate-z
                                      old-body-trans-rate-x
                                      old-body-trans-rate-y
                                      old-body-rotate-rate-z)
        (regulator)
:initable-instance-variables)


(defmethod (init joystick-command-regulator)
         ()
(setf body-trans-rate-x 0.0)
(setf body-trans-rate-y 0.0)
(setf body-rotate-rate-z 0.0)
(list body-trans-rate-x body-trans-rate-y body-rotate-rate-z))
```

```
(defmethod (regulate joystick-command-regulator)
           (joystick-command deceleration-factor)
  (if (<= deceleration-factor 0)
      (setf deceleration-factor 0.5))  ; remove effect of deceleration-factor.
  (let* ((d-const 0.5)
         (x (* (first  joystick-command) (/ d-const deceleration-factor)))
         (y (* (second joystick-command) (/ d-const deceleration-factor)))
         (r (* (third  joystick-command) (/ d-const deceleration-factor))))
    (setf body-trans-rate-x  (filter self x body-trans-rate-x))
    (setf body-trans-rate-y  (filter self y body-trans-rate-y))
    (setf body-rotate-rate-z (filter self r body-rotate-rate-z)))
    (if (< (abs body-trans-rate-x)  0.02)  (setf body-trans-rate-x  0.0))
    (if (< (abs body-trans-rate-y)  0.02)  (setf body-trans-rate-y  0.0))
    (if (< (abs body-rotate-rate-z) 0.005)  (setf body-rotate-rate-z  0.0))
    (list body-trans-rate-x body-trans-rate-y body-rotate-rate-z))


(defmethod (restore joystick-command-regulator)
           ()
  (setf body-trans-rate-x old-body-trans-rate-x)
  (setf body-trans-rate-y old-body-trans-rate-y)
  (setf body-rotate-rate-z old-body-rotate-rate-z)
  (list body-trans-rate-x body-trans-rate-y body-rotate-rate-z))


(defmethod (save joystick-command-regulator)
           ()
  (setf old-body-trans-rate-x body-trans-rate-x)
  (setf old-body-trans-rate-y body-trans-rate-y)
  (setf old-body-rotate-rate-z body-rotate-rate-z)
  (list body-trans-rate-x body-trans-rate-y body-rotate-rate-z))
```

```
; -*- Mode: LISP; Package: LEG; Syntax: Common-lisp; Base: 10 -*-

************************************************************

 state flavor definition

************************************************************

efflavor state(name next-state)
          ()
:initable-instance-variables)


efmethod (state-name state)
         ()
name)


efmethod (set-next-state state)
          (a-state)
(setf next-state a-state))




************************************************************

 sync-state flavor definition

************************************************************


efflavor sync-state((time 0) (del-t 0.1) time-out)
          (state)
:initable-instance-variables)


efmethod (change sync-state)
          ()
(setf time (+ time del-t))
(cond ((>= time time-out)
       (setf time 0)
       next-state)
      (t self)))


efmethod (get-time sync-state)
          ()
time)




************************************************************
```

```
;  async-state flavor definition
;
;*********************************************************

(defflavor async-state((command nil) (observation nil))
          (state)
  :initable-instance-variables)


(defmethod (change async-state)
          (given-command observed-event)
  (cond ((and (not observation)
              (equal given-command command))
         next-state)
        ((and (not command)
              (equal observed-event observation))
         next-state)
        (t self)))







;*********************************************************
;
;  state-machine flavor definition
;
;*********************************************************

(defflavor state-machine(state owner)
          ()
  :initable-instance-variables)


(defmethod (state-name state-machine)
          ()
  (state-name state))







;*********************************************************
;
;  control-state-machine flavor definition
;
;*********************************************************

(defflavor control-state-machine((command nil) (observation nil)
                                 contact-sensor executor)
          (state-machine)
  :initable-instance-variables)



(defmethod (init control-state-machine)
```

```
        (leg-name)
(if (member leg-name '(leg1 leg4 leg5))
    (init-control-machine self 'support)
    (init-control-machine self 'ready))
(setf contact-sensor (leg-contact-sensor owner))
(setf executor (leg-executor owner)))


efmethod (init-control-machine control-state-machine)
        (a-state-name)
internally used by init method
(let (return lift support contact descent advance ready)
  (setf return
        (make-instance 'sync-state
                       :name 'return :time-out 0.6))
  (setf lift
        (make-instance 'sync-state
                       :name 'lift :time-out 0.4
                       :next-state return))
  (setf support
        (make-instance 'async-state
                       :name 'support :command 'recover-command
                       :next-state lift))
  (setf contact
        (make-instance 'sync-state
                       :name 'contact :time-out 1.0
                       :next-state support))
  (setf descent
        (make-instance 'async-state
                       :name 'descent :observation 'contact-confirm
                       :next-state contact))
  (setf advance
        (make-instance 'sync-state
                       :name 'advance :time-out 0.6
                       :next-state descent))
  (setf ready
        (make-instance 'async-state
                       :name 'ready :command 'deploy-command
                       :next-state advance))

  (set-next-state return ready)


  (setf state (cond ((equal a-state-name (state-name ready))
                     ready)
                    ((equal a-state-name (state-name advance))
                     advance)
                    ((equal a-state-name (state-name descent))
                     descent)
                    ((equal a-state-name (state-name contact))
                     contact)
                    ((equal a-state-name (state-name support))
                     support)
                    ((equal a-state-name (state-name lift))
                     lift)
                    ((equal a-state-name (state-name return))
                     return)))
  )
)


defmethod (change control-state-machine :before)
```

```
           ()
  (cond ((typep state 'async-state)
         (if (sensing contact-sensor)
             (setf observation 'contact-confirm)
             (setf observation nil))
          )))


(defmethod (change control-state-machine)
           ()
  (cond ((typep state 'sync-state)
          (setf state (change state)))
         (t (setf state (change state command observation)))))


(defmethod (change control-state-machine :after)
           ()
  ; send command to executor with sync-state-time
  (send-command executor (state-name state))
  (if (typep state 'sync-state)
      (set-time executor (get-time state))
      (set-time executor nil)))



(defmethod (send-command control-state-machine)
           (a-command)
  (setf command a-command))
```

```
; -*- mode:lisp; syntax:common-lisp; Package: TERRAIN -*-

************************************************************

 display.globals

************************************************************

efvar eye-space nil)
efvar middle-of-screen nil)



efvar terrain-joystick)
efvar graph-terrain)
efvar graph-asv)


efvar terrain-sample nil)
efvar terrain-type "Random")
efvar terrain-angle nil)
efvar terrain-percent 0)
efvar terrain-seed 123)

************************************************************

 display.library

************************************************************



efun draw-to-earth (a-point)
(let ((draw-pt (make-displayable
         middle-of-screen
         (transform eye-space a-point))))
  (zl-user:draw-to
    (list (truncate (first draw-pt))
      (truncate (second draw-pt)))
    zl-user:*robot-window*)))



efun draw-to-earth-d (a-point)
(let ((draw-pt (make-displayable
         middle-of-screen
         (transform eye-space a-point))))
  (zl-user:draw-to-d
    (list (truncate (first draw-pt))
      (truncate (second draw-pt)))
    zl-user:*robot-window*)))



efun erase-to-earth (a-point)
(let ((draw-pt (make-displayable
         middle-of-screen
         (transform eye-space a-point))))
  (zl-user:erase-to
    (list (truncate (first draw-pt))
      (truncate (second draw-pt)))
    zl-user:*robot-window*)))
```

```
(defun eye-trans (eye-pt)
; eye-pt (radius alpha beta)
; eye:= orient*trans(0,0,-r)*rot(x,-beta)*rot(y,-alpha)*trans(-x,-y,-z)
; returns eye-space
; library : ident, transmat,rotate,matrixmult
  (let* ((orient (ident))
     (rot nil) (trans nil) (eye nil)
     (radius (first eye-pt)) (alpha (second eye-pt)) (beta (third eye-pt))
     (center-of-interest (list (/ (terrain-max-x graph-terrain) 2)
                               (/ (terrain-max-y graph-terrain) 2) 0)))
    (setf (aref orient 2 2) -1.0)
    (setf trans (transmat 0 0 (- radius)))
    (setf eye (matrixmult orient trans))
    (setf rot (rotatemat 'y-axis (- alpha)))
    (setf eye (matrixmult eye rot))
    (setf rot (rotatemat 'x-axis (- beta)))
    (setf eye (matrixmult eye rot))
    (setf trans (transmat (- (first center-of-interest))
              (- (second center-of-interest))
              (- (third center-of-interest))))
    (matrixmult eye trans)))



(defun make-displayable  (middle pt)
  (let ((scale   5000.0)
    (x (first pt)) (y (second pt)) (z (third pt)))
    (list (+ (* scale (/ x z)) (first middle))
      (+ (* scale (/ y z)) (second middle)))))


(defun move-to-earth (a-point)
  (let ((draw-pt (make-displayable
           middle-of-screen
           (transform eye-space a-point))))
    (zl-user:move-to
      (list (truncate (first draw-pt))
        (truncate (second draw-pt))))))







;*************************************************
;
;  joystick flavor definition
;
;*************************************************


(defflavor joystick((joy-x 0) (joy-y 0) (joy-r 0))
           ()
  :initable-instance-variables)



(defmethod (get-joy-value joystick)
           ()
  (let* ((key-value)
         (delta-x 0.2) (delta-y 0.1) (delta-r 0.01))
```

```
  (setf key-value (zl-user:get-keyboard-input))
  (cond ((equal key-value '#\f) (setf joy-x (+ joy-x delta-x)))
        ((equal key-value '#\b)  (setf joy-x (- joy-x delta-x)))
        ((equal key-value '#\r) (setf joy-y (- joy-y delta-y)))
        ((equal key-value '#\l) (setf joy-y (+ joy-y delta-y)))
        ((equal key-value '#\=)  (setf joy-r (- joy-r delta-r)))
        ((equal key-value '#\-)  (setf joy-r (+ joy-r delta-r))))
  (cond ((>= joy-x 2)  (setf joy-x 2))
        ((<= joy-x -2) (setf joy-x -2)))
  (cond ((>= joy-y 1)  (setf joy-y 1))
        ((<= joy-y -1) (setf joy-y -1)))
  (cond ((>= joy-r 0.1) (setf joy-r 0.1))
        ((<= joy-r -0.1) (setf joy-r -0.1)))
  (cond ((equal key-value '#\Circle) (setf joy-x 0)
         (setf joy-y 0) (setf joy-r 0)))
  (list joy-x joy-y joy-r (equal key-value '#\x))))


efmethod (reset joystick)
      ()
(setf joy-x 0)
(setf joy-y 0)
(setf joy-r 0))


etf terrain-joystick (make-instance 'joystick))



*********************************************************

 terrain flavor definition

*********************************************************


efflavor terrain((terrain-data (make-array '(49 49) :initial-element 0))
                 terrain-height-array terrain-height-list joystick
                 (cursor-x) (cursor-y) (max-x) (max-y))
         ()
:initable-instance-variables
:readable-instance-variables)


efmethod (create terrain)
         ()
 (init self)
 (modify self))



efmethod (get-height terrain)
         (a-pos-wrt-earth)
 range 0 =< x <= (first dimension-terrain-height)
       0 =< y <= (second dimension-terrain).
 (let* ((dimension-terrain-height (array-dimensions terrain-height-array))
        (x-min 0) (x-max (first dimension-terrain-height))
        (x (first a-pos-wrt-earth)))
   (if (or (< x x-min) (> x x-max))
       -1000
       (let* ((i-x (floor x))             ; get terrain x-index
```

```
                (x1 (if (< (- x i-x) 0.5) (- i-x 1) i-x))
                (x2 (if (< (- x i-x) 0.5) i-x (+ i-x 1)))
                (x1 (if (< x1 x-min) 0 x1))
                (x2 (if (>= x2 x-max) (- x-max 1) x2))
                (z1 (aref terrain-height-array x1))
                (z2 (aref terrain-height-array x2))
                (slope (- z2 z1))
                (del-x (- x x1)))
           (+ z1 (* slope del-x))))))



(defmethod (init terrain)
           ()
; globals : middle-of-screen, eye-space
  (let ((array-dims (array-dimensions terrain-data)))
    (setf max-x (first array-dims))
    (setf max-y (second array-dims))
    (setf cursor-x (floor (/ max-x 2)))
    (setf cursor-y (floor (/ max-y 2))))
  (zl-user:make-robot-window)
  (setf middle-of-screen
        (list (/ (zl-user:send zl-user:*robot-window* :inside-width) 2)
              (/ (zl-user:send zl-user:*robot-window* :inside-height) 2)))
  (setf eye-space (eye-trans (list 500 0 0)))
  (read-terrain-height self)
  (draw-terrain self eye-space)
  (display-cursor self)
  (zl-user:make-visible)
  (print "To modify this terrain use keyboard")
  (print "If ready, type any key and return")
  (read)
  (reset terrain-joystick))



(defmethod (in-side-of-whole-terrain terrain)
           (a-pos)
  (let ((dimension-terrain (array-dimensions terrain-data))
        (i-x (floor (first a-pos)))
        (i-y (floor (second a-pos))))
    (cond ((< i-x 0) nil)
          ((< i-y 0) nil)
          ((> i-x (- (first  dimension-terrain) 1)) nil)
          ((> i-y (- (second dimension-terrain) 1)) nil)
          (T))
    ))



(defmethod (modify terrain)
           ()
; external : eye-space
  (do ((radius 500) (alpha 0) (beta 0) (delta 0.0001)
       (joystick-value nil)
       (end-flag nil))
      (end-flag (reset joystick) (save-terrain self eye-space))
    (zl-user:make-visible)
    (setf joystick-value (get-joy-value joystick))
    (let ((x (first joystick-value))
          (y (second joystick-value))
          (r (third joystick-value))
          (fire (fourth joystick-value)))
```

```
    (erase-terrain self)
    (cond
      (fire (setf end-flag t))
      ((> x delta)      (setf alpha (+ alpha 0.05)))
      ((< x (- delta)) (setf alpha (- alpha 0.05)))
      ((> y delta)      (setf beta  (+ beta  0.05)))
      ((< y (- delta)) (setf beta  (- beta  0.05)))
      ((> r delta)      (setf radius (+ radius 10)))
      ((< r (- delta))  (setf radius (- radius 10))))
    (setf eye-space (eye-trans (list radius alpha beta)))
    (draw-terrain1 self eye-space))))



efmethod (permitted-cell terrain)
         (terrain-pos)
(let ((i-x (floor (first terrain-pos))) ; find terrain index
      (i-y (floor (second terrain-pos))))
  (if (in-side-of-whole-terrain self terrain-pos)
      (if (equal (aref terrain-data i-x i-y) 0) ; permitted
          t
          nil))))



efmethod (terrain-point terrain)
         (a-pos-wrt-earth)
(let* ((x (first a-pos-wrt-earth))
       (y (second a-pos-wrt-earth))
       (z (get-height self (list x y))))
  (list x y z)))



etf graph-terrain (make-instance 'terrain
                                 :joystick terrain-joystick))



efun init-terrain()
called to create terrain
(create graph-terrain))



**********************************************************

 terrain.display-terrain

**********************************************************



efmethod (display-cursor terrain)
         ()
(make-all-permitted self)
(tv:choose-variable-values
  '("Choose terrain type"
     (terrain-type "Terrain type" :choose ("Random"  "Manual"))))
(if (equal terrain-type "Random")
    (random-terrain self)
    (do ((joy-data nil) (x nil) (y nil) (r nil) (fire nil)
```

```
       (exit-flag nil))
      (exit-flag (erase-cursor self (list cursor-x cursor-y)))
    (zl-user:make-visible)
    (setf joy-data (get-joy-value joystick))
    (setf x (- (second joy-data))) (setf y (first joy-data))
    (setf r (third joy-data)) (setf fire (fourth joy-data))
    (erase-cursor self (list cursor-x cursor-y))
    (cond
      (fire (setf exit-flag t))
      ((> x 0) (setf cursor-x (+ cursor-x 1)) (if (> cursor-x max-x)
                              (setf cursor-x max-x)))
      ((< x 0) (setf cursor-x (- cursor-x 1)) (if (< cursor-x 0)
                              (setf cursor-x 0)))
      ((> y 0) (setf cursor-y (+ cursor-y 1)) (if (> cursor-y max-y)
                              (setf cursor-y max-y)))
      ((< y 0) (setf cursor-y (- cursor-y 1)) (if (< cursor-y 0)
                              (setf cursor-y 0)))
      ((< r 0) (setf (aref terrain-data cursor-x cursor-y) 1))
      ((> r 0) (setf (aref terrain-data cursor-x cursor-y) 1)))
    (draw-cursor self (list cursor-x cursor-y))
    (draw-obstacles self)
    (reset joystick))))



(defmethod (draw-terrain terrain)
           (eye-space)
; external function: \display.library\move-to-earth, draw-to-earth
  (dotimes (x (+ max-x 1))
    (move-to-earth (list x 0 (aref terrain-height-array x)))
    (draw-to-earth (list x max-x (aref terrain-height-array x))))
  (dotimes (y (+ max-y 1))
    (move-to-earth (list 0 y 0))
    (dotimes (x (+ max-x 1))
      (draw-to-earth (list x y (aref terrain-height-array x))))))


(defmethod  (draw-terrain1 terrain)
            (eye-space)
; external function: \display.library\move-to-earth, draw-to-earth
  (do ((xs (list 0 max-x) (cdr xs))
       (x nil))
      ((null xs))
    (setf x (car xs))
    (move-to-earth (list x 0 (aref terrain-height-array x)))
    (draw-to-earth (list x max-x (aref terrain-height-array x))))
  (do ((ys (list 0 max-y) (cdr ys))
       (y nil))
      ((null ys))
    (setf y (car ys))
    (move-to-earth (list 0 y 0))
    (dotimes (x (+ max-x 1))
      (draw-to-earth (list x y (aref terrain-height-array x))))))


(defmethod (erase-obstacles terrain)
           ()
; externals : terrain
; external function: \display.library\move-to-earth, draw-to-earth
  (dotimes (i (first (array-dimensions terrain-data)))
    (dotimes (j (second (array-dimensions terrain-data)))
      (cond ((equal 1 (aref terrain-data i j))
         (move-to-earth (list i j))
         (erase-to-earth (list (+ i 1) (+ j 1)))
```

```
      (move-to-earth (list (+ i 1) j))
      (erase-to-earth (list i (+ j 1))))))))


fmethod (erase-terrain terrain)
        ()
tv:sheet-force-access (zl-user:*robot-window*)
 (send zl-user:*robot-window* :clear-window)))



fmethod (make-all-permitted terrain)
        ()
dotimes (i max-x)
 (dotimes (j max-y)
   (setf (aref terrain-data i j) 0))))


fmethod (read-terrain-height terrain)
        ()
(tv:choose-variable-values
  '("Choose a terrain"
    (terrain-sample "Terrain sample" :choose ("Sample" "Angle" "Custom"))))
setf terrain-sample "Sample")
cond ((equal terrain-sample "Sample")
      (setf terrain-height-list '((19 0) (25 1) (35 1.5))))
     ((equal terrain-sample "Angle")
      (tv:choose-variable-values
        '((terrain-angle "Angle in degree" :number))
        ':label "Slope of terrain")
      (let* ((angle (* pi (/ terrain-angle 180)))
             (max (* 20 (tan angle))))
        (setf terrain-height-list
              (list '(20 0)
                    (list 40 max)
                    ))))
     (t (print "Please input terrain height.")
        (setf terrain-height-list (read))))
setf terrain-height-array (make-array (+ max-x 1)))
let* ((x1 0) (z1 0) (a-pair) (zz 0)
      (x2 (first (car terrain-height-list)))
      (z2 (second (car terrain-height-list)))
      (slope (/ (- z2 z1) (- x2 x1))))
 (setf terrain-height-list (cdr terrain-height-list))
 (dotimes (i (+ max-x 1))
   (setf zz (+ (* slope (- i x1)) z1))
   (cond ((equal x2 i)
          (setf x1 x2)
          (cond ((setf a-pair (car terrain-height-list))
                 (setf terrain-height-list (cdr terrain-height-list))
                 (setf x2 (first a-pair))
                 (setf z2 (second a-pair))
                 (setf z1 zz)
                 (setf slope (/ (- z2 z1) (- x2 x1))))
                (T (setf slope 0) (setf z1 zz)))))
   (setf (aref terrain-height-array i) zz))))


fmethod (save-terrain terrain)
        (eye-space)
```

```
  (draw-obstacles self)
  (draw-terrain self eye-space)
  (zl-user:save-terrain-to-terrain-buffer))




;*********************************************************

;  terrain.display-cursor

;*********************************************************


(defmethod (draw-cursor terrain)
           ( position )
  (let* ((x (first position))
         (y (second position))
         (p1 (list (+ x 0.2) (+ y 0.2) 0))
         (p2 (list (+ x 0.8) (+ y 0.2) 0))
         (p3 (list (+ x 0.8) (+ y 0.8) 0))
         (p4 (list (+ x 0.2) (+ y 0.8) 0))
         (points (list p2 p3 p4 p1)))
    (move-to-earth p1)
    (do ((points points (cdr points)))
        ((null points) 'done-draw-cursor)
      (draw-to-earth (car points)))))



(defmethod (draw-obstacles terrain)
           ()
  (dotimes (i (first (array-dimensions terrain-data)))
    (dotimes (j (second (array-dimensions terrain-data)))
      (cond ((equal 1 (aref terrain-data i j))
             (move-to-earth
               (list i j (aref terrain-height-array i)))
             (draw-to-earth
               (list (+ i 1) (+ j 1) (aref terrain-height-array (+ i 1))))
             (move-to-earth
               (list (+ i 1) j (aref terrain-height-array (+ i 1))))
             (draw-to-earth
               (list i (+ j 1) (aref terrain-height-array i))))))))



(defmethod (erase-cursor terrain)
           ( position )
  (let* ((x (first position))
         (y (second position))
         (p1 (list (+ x 0.2) (+ y 0.2) 0))
         (p2 (list (+ x 0.8) (+ y 0.2) 0))
         (p3 (list (+ x 0.8) (+ y 0.8) 0))
         (p4 (list (+ x 0.2) (+ y 0.8) 0))
         (points (list p2 p3 p4 p1)))
    (move-to-earth p1)
    (do ((points points (cdr points)))
        ((null points) 'done-erase-cursor)
      (erase-to-earth (car points)))))
```

```
efmethod (random-terrain terrain)
         ()
(let ((a 43411) (b 17) (c 640001) (x nil))
  (tv:choose-variable-values
    '((terrain-percent "Obstacles in percentage" :number)
      (terrain-seed "Random number seed" :number))
    ':label "How much obstacles on the terrain in percentage? ")
  (setf x terrain-seed)
  (dotimes (i max-x)
    (dotimes (j max-y)
  (if (< (/ (setf x (mod (+ (* a x) b) c)) c) (/ terrain-percent 100))
      (setf (aref terrain-data i j) 1)))))
(draw-obstacles self)
(zl-user: make-visible))




**********************************************************

 graph-vehicle flavor definition

**********************************************************

efflavor graph-vehicle((vehicle-points (make-array 28))
                       (body-points (make-array 10))
                       (polygons (make-array 13))
                       (numpolys nil)
                       (vertices (make-array 100)))
         ()
:initable-instance-variables)



efmethod (init-data graph-vehicle)
         ()
(read-vehicle-data self))  ; read data from disk


efmethod (display graph-vehicle)
         (a-H foot-positions)
(tv:sheet-force-access (zl-user:*robot-window*)
  (send zl-user:*robot-window* :clear-window))
(zl-user:copy-terrain-to-robot-window)
(body-pento-wrt-earth self a-H foot-positions)
(draw-vehicle self vehicle-points)
(zl-user:make-visible))



efmethod (read-vehicle-data graph-vehicle)
         ()
global variables : vehicle-points, polygons, numpolys, vertices
format of file : num-of-points    num-of-polygons
                 ( num a-vehicle-point) ....
                 ( num-of-vertices  vertices-number-of-a-polygon)...
```

```
  (let* ((vehicle-file (open "object:vehicle.data"))
     (numpts (read vehicle-file))
     (numvtces 0) (a-polygon nil))
    (setf numpolys (read vehicle-file))
    (dotimes (i numpts)
      (setf (aref vehicle-points i) (cdr (read vehicle-file))))
    (dotimes (i 10)
      (setf (aref body-points i) (aref vehicle-points i)))
    (dotimes (i numpolys)
      (setf a-polygon (read vehicle-file))
      (setf (aref polygons i) (list numvtces (car a-polygon)))
      (do ((a-polygon-vertices (cdr a-polygon) (cdr a-polygon-vertices))
       (j 0 (+ j 1)))
      ((null a-polygon-vertices))
    (setf (aref vertices  (+ numvtces j))
          (- (first a-polygon-vertices) 1)))
      (setf numvtces (+ numvtces (car a-polygon))))
    (close vehicle-file)))



(setf graph-asv (make-instance 'graph-vehicle))



;*************************************************************
;
;  graph-vehicle.display
;
;*************************************************************



(defmethod (body-pento-wrt-earth graph-vehicle)
           ( a-H foot-positions )
; library : transform
  (let ((s1 0.6616) (s2 0.945) (s3 3.308) (l 0.8133) (m 1.0467)
    (hipx-list '(5.1667 5.1667 0.0 0.0 -4.9167 -4.9167))
    (hipy-list '(1.62 -1.62 1.62 -1.62 1.62 -1.62))
    (sign1-list '(1 -1  1 -1  1 -1))
    (sign2-list '(1  1  1  1 -1 -1)))
   (transform-body-points self a-H body-points)
    (do ((positions foot-positions (cdr positions))
     (hipx-list hipx-list (cdr hipx-list))
     (hipy-list hipy-list (cdr hipy-list))
     (sign1-list sign1-list (cdr sign1-list))
     (sign2-list sign2-list (cdr sign2-list))
     (i 0 (+ i 1)))
    ((null positions) nil)
      (let* ((foot-pos (car positions))
         (hipx (car hipx-list)) (hipy (car hipy-list))
         (sign1 (car sign1-list)) (sign2 (car sign2-list))
         (px (- (first foot-pos) hipx))
         (py (- (second foot-pos) hipy))
         (pz (third foot-pos))
         (theta (vehicle-theta py pz m sign1))
         (dm (vehicle-dm px sign2))
         (dl (vehicle-dl py pz m l))
         (top-pos nil) (knee-pos nil))
    (setf top-pos
          (transform a-H
             (vehicle-top-pos hipx hipy m l dl theta sign1)))
    (setf knee-pos
          (transform a-H
```

```
          (vehicle-knee-pos hipx hipy m l s1 s2 s3
                    dl dm theta sign1 sign2)))
  (setf foot-pos (transform a-H foot-pos))
  (setf (aref vehicle-points (+ 10 (* 3 i)))
        top-pos)
  (setf (aref vehicle-points (+ 11 (* 3 i)))
        knee-pos)
  (setf (aref vehicle-points (+ 12 (* 3 i)))
        foot-pos)))))



efmethod (draw-vehicle graph-vehicle)
         ( vehicle-points )
global variables :  polygons, numpolys, vertices
(dotimes (i numpolys)
  (let ((start (first (aref polygons i)))
    (num-vertices (second (aref polygons i))))
    (move-to-earth (aref vehicle-points
              (aref vertices start)))
   (dotimes (j num-vertices)
     (draw-to-earth-d (aref vehicle-points
               (aref vertices (+ start j)))))
   )))




************************************************************

 graph-vehicle.display.body-pento-wrt-earth

************************************************************


efmethod (transform-body-points graph-vehicle)
         (a-H body-points)
globals : vehicle-points
library : transform
(dotimes (i 10)
  (setf (aref vehicle-points i)
    (transform a-H (aref body-points i)))))



efun vehicle-dl (py pz m l)
(/ (- (sqrt (+ (* py py) (* pz pz) (- (* m m)))) l)
   4))



efun vehicle-dm (px sign2)
(* sign2 (/ px 5)))



efun vehicle-knee-pos (hipx hipy m l s1 s2 s3
                dl dm theta sign1 sign2)
(let* ((numer (+ (* s1 s1) (- (* s2 s2)) (* dl dl) (* dm dm)))
   (denom (* 2 s1 (sqrt (+ (* dl dl) (* dm dm)))))
   (beta (acos (/ numer denom)))
   (alpha (- (/ pi 2) (atan dm dl) beta))
```

```
     (sina (sin alpha)) (cosa (cos alpha))
     (sint (sin theta)) (cost (cos theta))
     (temp (- (* s3 sina) (- dl l)))
     (xk (+ (* sign2 s3 cosa) hipx))
     (yk (+ (* sign1 (+ (* temp sint) (* m cost))) hipy))
     (zk (- (+ (* temp cost) (* m sint)))))
    (list xk yk zk)))


(defun vehicle-theta (py pz m sign1)
  (let* ((angle1 (atan (* sign1 py) (* -1 pz)))
     (angle2 (atan m (sqrt (+ (* py py)
                  (* pz pz)
                  (- (* m m)))))))
    (- angle1 angle2)))


(defun vehicle-top-pos (hipx hipy m l dl theta sign1)
  (let* ((xt hipx)
     (l-dl (- l dl))
     (sina (sin theta))
     (cosa (cos theta))
     (yt (+ (* sign1 (+ (* m cosa) (* l-dl sina))) hipy))
     (zt (- (* m sina) (* l-dl cosa))))
    (list xt yt zt)))
```

```
; -*- Mode: LISP; Package: LEG; Syntax: Common-lisp; Base: 10 -*-

****************************************************************

 executor flavor definition

****************************************************************


efflavor executor
      (leg-pos-wrt-body desired-foothold-pos-wrt-earth
                        time command owner sensor (lift-height 1.4)
                        (T1 0.6) (T2 1.0) (T3 0.4) (T4 0.6)
                        (planned-contact-time 0.4) self-time
                        (sampling-time 0.1) ready-pos
                        H1 inv-H1 body-trans-rate1 body-rotate-rate1)
      ()
:initable-instance-variables)


efmethod (set-desired-pos executor)
         (a-pos)
(setf desired-foothold-pos-wrt-earth a-pos))



efmethod (get-desired-pos executor)
         ()
desired-foothold-pos-wrt-earth)


efmethod (send-command executor)
         (a-command)
(setf command a-command))


efmethod (set-time executor)
         (a-time)
(setf time a-time))


efmethod (leg-pos-wrt-body executor)
         ()
leg-pos-wrt-body)


efmethod (move executor)
         (H inv-H body-trans-rate body-rotate-rate)
(setf H1 H)
(setf inv-H1 inv-H)
(setf body-trans-rate1 body-trans-rate)
(setf body-rotate-rate1 body-rotate-rate)
(cond ((equal command 'ready)
       (move-in-ready self))
      ((equal command 'advance)
       (move-in-advance self))
      ((equal command 'descent)
       (move-in-descent self))
      ((equal command 'contact)
```

```
         (move-in-contact self))
        ((equal command 'support)
         (move-in-support self))
        ((equal command 'lift)
         (move-in-lift self))
        ((equal command 'return)
         (move-in-return self))
        )
  )


(defmethod (move-in-contact executor)
           ()
  (let ((leg-velocity-wrt-body (find-velocity-wrt-body self)))
    (setf leg-pos-wrt-body
          (vectadd (magvect sampling-time leg-velocity-wrt-body)
                   leg-pos-wrt-body))))



(defmethod (find-velocity-wrt-body executor)
           ()
; returns foot-velocity-wrt-body
; velocity = - ( body-trans-rate + body-rotate-rate X leg-pos )
; globals v : body-trans-rate1, body-rotate-rate1
; lib : vectsub, vectadd, crossprod
  (vectsub '(0 0 0)
           (vectadd body-trans-rate1
                    (crossprod body-rotate-rate1 leg-pos-wrt-body))))



(defmethod (move-in-advance executor)
           ()
  (let ((desired-pos (desired-advance-pos-wrt-body self))
        (dt (- T1 time)))
    (move-del self desired-pos leg-pos-wrt-body dt))
  (setf self-time 0.0))



(defmethod (desired-advance-pos-wrt-body executor)
           ()
; a-pos is desired-stepping-pos-wrt-earth
; returns desired-pos-wrt-body in deploy state
; global variable : H1, inv-H1
; global function : to-earth-transform, to-body-transform, find-terrain-hegiht
  (let* ((desired-pos-wrt-earth desired-foothold-pos-wrt-earth)
         (terrain-height (third (terrain-point owner desired-pos-wrt-earth)))
         (desired-pos-height-wrt-earth (+ terrain-height lift-height))
         (pos-wrt-earth (list (first  desired-pos-wrt-earth)
                              (second desired-pos-wrt-earth)
                               desired-pos-height-wrt-earth)))
    (to-body-transform inv-H1 pos-wrt-earth)))



(defmethod (move-in-descent executor)
           ()
; global function : to-body-transform
; global variables : inv-H1
  (let ((dt (- planned-contact-time self-time)))
    (if (< dt 0.05)
        (setf leg-pos-wrt-body (to-body-transform
```

```
                                inv-H1 desired-foothold-pos-wrt-earth))
      (move-del self
        (to-body-transform inv-H1 desired-foothold-pos-wrt-earth)
        leg-pos-wrt-body dt))))



efmethod (move-in-descent executor :after)
         ()
(setf self-time (+ self-time sampling-time)))



efmethod (move-del executor)
         (desired-pos present-pos dt)
set new leg-pos depending on the arguments
lib : vectadd, magvect
(if (< dt 0.05)
    (setf leg-pos-wrt-body desired-pos)
    (let* ((inv-time-diff (/ 1 dt))
           (del (vectsub desired-pos present-pos))
           (velocity (magvect inv-time-diff del)))
      (setf leg-pos-wrt-body
            (vectadd present-pos (magvect sampling-time velocity))))))



efmethod (move-in-lift executor)
         ()
(let* ((dt (- T3 time))
       (desired-pos (lift-pos-desired self))
       (z (third desired-pos)))
  (move-del self desired-pos leg-pos-wrt-body  dt)
  (setf ready-pos
          (list (first ready-pos) (second ready-pos) z))))



efmethod (lift-pos-desired executor)
eturens position-wrt-body which will be at the end of lift state.
global f : to-body-transform,
global v : inv-H1
         ()
(let* ((leg-pos-wrt-earth (to-earth-transform H1 leg-pos-wrt-body))
       (desired-height (+ lift-height (third (terrain-point owner leg-pos-wrt-earth)))))
  (to-body-transform inv-H1 (list (first  leg-pos-wrt-earth)
                                  (second leg-pos-wrt-earth)
                                  desired-height))))



efmethod (move-in-ready executor)
         ()
(setf leg-pos-wrt-body ready-pos))



efmethod (move-in-return executor)
         ()
Modifying leg-pos-z is redundent but it can correct disturbance by itself.
(let ((dt (- T4 time))
      (desired-pos ready-pos))
  (move-del self desired-pos leg-pos-wrt-body dt)))
```

```
(defmethod (move-in-support executor)
           ()
; globals : body-trans-rate1, body-rotate-rate1
; lib : vectadd, magvect
; In general terrain, leg-pos-z should be updated by real terrain height.
  (let ((leg-velocity-wrt-body (find-velocity-wrt-body self)))
    (setf leg-pos-wrt-body
          (vectadd (magvect sampling-time leg-velocity-wrt-body)
                   leg-pos-wrt-body))))



(defmethod (init executor)
           (leg-name init-H)
  (setf sensor (leg-contact-sensor owner))
  (let ((x (aref init-H 0 3))
        (y (aref init-H 1 3))
        (z (aref init-H 2 3)))
    (cond ((equal leg-name 'leg1)
           (setf ready-pos '( 5  3 -4))
           (setf leg-pos-wrt-body (list  6  3 (- z)))
           (setf desired-foothold-pos-wrt-earth (list (+ x 6) (+ y 3) 0)))
          ((equal leg-name 'leg2)
           (setf ready-pos '( 5 -3 -4))
           (setf leg-pos-wrt-body (list  5 -3 (- z)))
           (setf desired-foothold-pos-wrt-earth (list (+ x 5) (- y 3) 0)))
          ((equal leg-name 'leg3)
           (setf ready-pos '( 0  3 -4))
           (setf leg-pos-wrt-body (list  0  3 (- z)))
           (setf desired-foothold-pos-wrt-earth (list (+ x 0) (+ y 3) 0)))
          ((equal leg-name 'leg4)
           (setf ready-pos '( 0 -3 -4))
           (setf leg-pos-wrt-body (list  0 -3 (- z)))
           (setf desired-foothold-pos-wrt-earth (list (+ x 0) (- y 3) 0)))
          ((equal leg-name 'leg5)
           (setf ready-pos '(-5  3 -4))
           (setf leg-pos-wrt-body (list -5  3 (- z)))
           (setf desired-foothold-pos-wrt-earth (list (- x 5) (+ y 3) 0)))
          ((equal leg-name 'leg6)
           (setf ready-pos '(-5 -3 -4))
           (setf leg-pos-wrt-body (list -5 -3 (- z)))
           (setf desired-foothold-pos-wrt-earth (list (- x 5) (- y 3) 0)))))
  )
```

```
; -*- Mode: LISP; Syntax: Common-lisp; Package: LEG -*-


efflavor foothold-finder(sixteen-footholds
                         four-lines tkm-calculator
                         (no-cell-available-flag nil)
                         (TKM-margin 0.4) owner)
         ()
:initable-instance-variables)


efmethod (init foothold-finder)
         (leg-name)
(cond ((equal leg-name 'leg1)
       (setf sixteen-footholds
             '(( 7.3  4.3) ( 7.3  3.3) ( 7.3  2.3) ( 7.3  1.3)
               ( 6.3  4.3) ( 6.3  3.3) ( 6.3  2.3) ( 6.3  1.3)
               ( 5.3  4.3) ( 5.3  3.3) ( 5.3  2.3) ( 5.3  1.3)
               ( 4.3  4.3) ( 4.3  3.3) ( 4.3  2.3) ( 4.3  1.3)))
       (setf four-lines
             '(((0  0.3420 -0.9397) ( 8.0832  2.7339 0))
               ((0 -0.3420 -0.9397) ( 8.0832  2.7339 0))
               ((0 -0.3420 -0.9397) ( 3.4167  2.7339 0))
               ((0  0.3420 -0.9397) ( 3.4167  2.7339 0)))))
      ((equal leg-name 'leg2)
       (setf sixteen-footholds
             '(( 7.3 -4.3) ( 7.3 -3.3) ( 7.3 -2.3) ( 7.3 -1.3)
               ( 6.3 -4.3) ( 6.3 -3.3) ( 6.3 -2.3) ( 6.3 -1.3)
               ( 5.3 -4.3) ( 5.3 -3.3) ( 5.3 -2.3) ( 5.3 -1.3)
               ( 4.3 -4.3) ( 4.3 -3.3) ( 4.3 -2.3) ( 4.3 -1.3)))
       (setf four-lines
             '(((0  0.3420 -0.9397) ( 8.0832 -2.7339 0))
               ((0 -0.3420 -0.9397) ( 8.0832 -2.7339 0))
               ((0 -0.3420 -0.9397) ( 3.4167 -2.7339 0))
               ((0  0.3420 -0.9397) ( 3.4167 -2.7339 0)))))
      ((equal leg-name 'leg3)
       (setf sixteen-footholds
             '(( 1.5  4.3) ( 1.5  3.3) ( 1.5  2.3) ( 1.5  1.3)
               ( 0.5  4.3) ( 0.5  3.3) ( 0.5  2.3) ( 0.5  1.3)
               (-0.5  4.3) (-0.5  3.3) (-0.5  2.3) (-0.5  1.3)
               (-1.5  4.3) (-1.5  3.3) (-1.5  2.3) (-1.5  1.3)))
       (setf four-lines
             '(((0  0.3420 -0.9397) ( 2.2915  2.7339 0))
               ((0 -0.3420 -0.9397) ( 2.2915  2.7339 0))
               ((0 -0.3420 -0.9397) (-2.2915  2.7339 0))
               ((0  0.3420 -0.9397) (-2.2915  2.7339 0)))))
      ((equal leg-name 'leg4)
       (setf sixteen-footholds
             '(( 1.5 -4.3) ( 1.5 -3.3) ( 1.5 -2.3) ( 1.5 -1.3)
               ( 0.5 -4.3) ( 0.5 -3.3) ( 0.5 -2.3) ( 0.5 -1.3)
               (-0.5 -4.3) (-0.5 -3.3) (-0.5 -2.3) (-0.5 -1.3)
               (-1.5 -4.3) (-1.5 -3.3) (-1.5 -2.3) (-1.5 -1.3)))
       (setf four-lines
             '(((0  0.3420 -0.9397) ( 2.2915 -2.7339 0))
               ((0 -0.3420 -0.9397) ( 2.2915 -2.7339 0))
               ((0 -0.3420 -0.9397) (-2.2915 -2.7339 0))
               ((0  0.3420 -0.9397) (-2.2915 -2.7339 0)))))
      ((equal leg-name 'leg5)
       (setf sixteen-footholds
             '((-4.0  4.3) (-4.0  3.3) (-4.0  2.3) (-4.0  1.3)
               (-5.0  4.3) (-5.0  3.3) (-5.0  2.3) (-5.0  1.3)
               (-6.0  4.3) (-6.0  3.3) (-6.0  2.3) (-6.0  1.3)
               (-7.0  4.3) (-7.0  3.3) (-7.0  2.3) (-7.0  1.3)))
```

```
         (setf four-lines
               '(((0  0.3420 -0.9397) (-3.3332  2.7339 0))
                 ((0 -0.3420 -0.9397) (-3.3332  2.7339 0))
                 ((0 -0.3420 -0.9397) (-7.8332  2.7339 0))
                 ((0  0.3420 -0.9397) (-7.8332  2.7339 0)))))
        ((equal leg-name 'leg6)
         (setf sixteen-footholds
               '((-4.0 -4.3) (-4.0 -3.3) (-4.0 -2.3) (-4.0 -1.3)
                 (-5.0 -4.3) (-5.0 -3.3) (-5.0 -2.3) (-5.0 -1.3)
                 (-6.0 -4.3) (-6.0 -3.3) (-6.0 -2.3) (-6.0 -1.3)
                 (-7.0 -4.3) (-7.0 -3.3) (-7.0 -2.3) (-7.0 -1.3)))
         (setf four-lines
               '(((0  0.3420 -0.9397) (-3.3332 -2.7339 0))
                 ((0 -0.3420 -0.9397) (-3.3332 -2.7339 0))
                 ((0 -0.3420 -0.9397) (-7.8332 -2.7339 0))
                 ((0  0.3420 -0.9397) (-7.8332 -2.7339 0)))))

        )
  (setf tkm-calculator (leg-tkm-calculator owner))
  )



(defmethod (find-foothold foothold-finder)
           (H6 inv-H6 body-trans-rate10 body-rotate-rate10
               estimated-support-plane)
; returns ((max-foothold max-tkm) (foothold-list) (tkm-list))
; all points are wpt body coordinate system.
  (let* ((estimated-support-plane-wrt-body
          (plane-transform estimated-support-plane H6))
         (four-points (four-points-on-support-plane
                       self
                       four-lines estimated-support-plane-wrt-body))
         (possible-footholds (get-possible-footholds
                              self
                              (estimate-footholds
                                self
                                four-points estimated-support-plane-wrt-body)
                              H6 inv-H6)))
    (get-foothold-with-max-TKM
      self
      possible-footholds H6
      body-trans-rate10 body-rotate-rate10)))




;"*****************************************************"

; foothold-finder.find-foothold

;"*****************************************************"



(defmethod (estimate-footholds foothold-finder)
           (four-points-wrt-body estimated-support-plane-wrt-body)
; returns estimate-footholds-wrt-body
  (do* ((footholds sixteen-footholds (cdr footholds))
```

```
      (out-footholds nil)
      (a-foothold nil))
     ((null footholds)
      (get-points-on-support-plane out-footholds estimated-support-plane-wrt-body))
  (setf a-foothold (car footholds))
  (if (in-side-of-polygon a-foothold
                          (pick-two-dimensions four-points-wrt-body))
      (setf out-footholds (cons a-foothold out-footholds)))))



efmethod (four-points-on-support-plane foothold-finder)
         (four-lines estimated-support-plane-wrt-body)
returns four points which are intersected by four-lines on
estimated-support-plane-wrt-body
math lib: plane-intersection
(do* ((lines four-lines (cdr lines))
      (points nil))
     ((null lines) points)
  (setf points (cons (plane-intersection (car lines)
                                         estimated-support-plane-wrt-body)
                     points))))



efmethod (get-foothold-with-max-TKM foothold-finder)
         (possible-footholds H
                             body-trans-rate body-rotate-rate)
returns ((max-foothold max-tkm) (foothold-list) (tkm-list))
sets no-cell-available-flag
real-footholds is really possible footholds
  (do ((footholds possible-footholds (cdr footholds))
       (max-foothold nil) (a-foothold nil) (TKM-list nil) (a-TKM nil)
       (real-footholds nil) (max-TKM -100.0))
      ((null footholds)
       (setf no-cell-available-flag (< max-TKM TKM-margin))
       (if (>= max-TKM TKM-margin)
           (make-output-form
            max-foothold max-TKM real-footholds TKM-list H)
           nil))
    (setf a-foothold (car footholds))
    (setf a-TKM (find-tkm tkm-calculator
                          a-foothold body-trans-rate body-rotate-rate))
    (if a-TKM
        (progn (setf TKM-list (cons a-TKM TKM-list))
               (setf real-footholds (cons a-foothold real-footholds))
               (if (> a-TKM max-TKM)
                   (progn (setf max-TKM a-TKM)
                          (setf max-foothold a-foothold)))))))



efmethod (get-possible-footholds foothold-finder)
         (estimated-footholds H inv-H)
returns possible-footholds wrt body
(to-body-transform inv-H
                   (find-possible-footholds self
                       (to-earth-transform H estimated-footholds))
))
```

```
;"*****************************************************"

;  foothold-finder.estimate-foothold

;"*****************************************************"



(defun check-polarity (point1 point2 point3)
  (let* ((vect1 (vectsub point2 point1))
         (vect2 (vectsub point3 point1)))
    (if (not (third vect1))
        (progn (setf vect1 (reverse (cons 0 (reverse vect1))))
               (setf vect2 (reverse (cons 0 (reverse vect2))))))
    (crossprod vect1 vect2)))



(defun get-points-on-support-plane (points estimated-support-plane-wrt-body)
; returns intersection points with support plane in z-body direction.
; math lib: plane-intersection
  (do* ((points points (cdr points))
        (out-points nil))
       ((null points) out-points)
    (setf out-points (cons (plane-intersection
                             (make-line-to-get-point-on-support-plane
                                 (car points))
                             estimated-support-plane-wrt-body) out-points))))



(defun in-side-of-polygon (a-point polygon-points)
; polygon-points must be convext-polygon and in order & two dimensional points.
  (do* ((first-points  polygon-points (cdr first-points))
        (second-points (reverse (cons (car first-points)
                                      (reverse (cdr first-points))))
                       (cdr second-points))
        (signs nil) (first-point nil) (second-point nil))
       ((null first-points) (same-polarity signs))
    (setf first-point  (car first-points))
    (setf second-point (car second-points))
    (setf signs (cons (check-polarity first-point second-point a-point)
                      signs))))



(defun make-line-to-get-point-on-support-plane (a-point)
; a-point is two dimensional point.
; returns a-line ((z-direction) (a-point -100))
  (list '(0 0 1) (list (first a-point) (second a-point) -100)))



(defun pick-two-dimensions (points)
  (if (listp (first points))
      (do* ((points points (cdr points))       ; more than one point case
            (a-point nil)
            (out-points nil))
           ((null points) out-points)
        (setf a-point (car points))
        (setf out-points (cons (list (first a-point) (second a-point))
                               out-points)))
      (list (first points) (second points))))  ; one point case
```

```
efun same-polarity (signs)
(do ((signs (cdr signs) (cdr signs))
     (first-sign (plusp (third (car signs))))
     (same T))
    ((null signs) same)
  (if (not (equal first-sign (plusp (third (car signs)))))
      (setf same nil))))




**********************************************************"

 foothold-finder.find-foothold.get-foothold-with-MAX-tkm

**********************************************************"


efun make-output-form
     (max-foothold max-TKM possible-footholds TKM-list H)
output-form : ((foothold-with-max-tkm tkm)
               (leg-projected-permitted-footholds)
               (leg-projected-TKM-list))
output footholds are in earth coordinate.
math lib : to-earth-transform
(list (list (to-earth-transform H max-foothold) max-TKM)
      (to-earth-transform H possible-footholds)
      TKM-list))




**********************************************************"

 foothold-finder.select-foothold.get-possible-foothold

**********************************************************"


efmethod (find-possible-footholds foothold-finder)
         (estimated-footholds-wrt-earth)
returns possible-footholds-wrt-earth
use vision
(do* ((footholds estimated-footholds-wrt-earth (cdr footholds))
      (a-foothold nil) (t-cell nil) (out-footholds nil))
     ((null footholds) (unique-footholds-only out-footholds))
  (setf a-foothold (car footholds))
  (setf t-cell (get-center-of-digitized-terrain-cell a-foothold))
  (if (permitted-cell owner t-cell)
      (setf out-footholds
            (cons (terrain-point owner t-cell)
                  out-footholds)))))


efun get-center-of-digitized-terrain-cell (a-foothold)
```

```
; cell resolution is 1 foot by 1 foot
  (list (+ (floor (first  a-foothold)) 0.5)
        (+ (floor (second a-foothold)) 0.5)))



(defun unique-footholds-only (mixed-footholds)
  (do* ((footholds mixed-footholds (cdr footholds))
        (out-footholds nil)
        (a-foothold nil))
       ((null footholds) out-footholds)
    (setf a-foothold (car footholds))
    (if (not (member a-foothold out-footholds :test 'equal))
        (setf out-footholds (cons a-foothold out-footholds)))))
```

```
; -*- mode:lisp; syntax:zetalisp; Package: USER -*-

*******************************************************

 low level graph routines

*******************************************************



efvar *robot-display-window* nil)
efvar *robot-display-window-array* nil)
efvar *robot-window* nil)
efvar *robot-window-array* nil)
efvar *robot-window-width* nil)
efvar *robot-window-height* nil)
efvar *terrain-buffer* nil)
efvar *terrain-buffer-array* nil)
efvar *max-y* nil)
efvar *start-point* nil)


efun kill-all-windows()
(send *robot-display-window* :kill)
(send *robot-window* :kill)
(send *terrain-buffer* :kill))


efun copy-terrain-to-robot-window ()
(tv:sheet-force-access (*robot-window*)
  (send *robot-window* :bitblt
    tv:alu-ior *robot-window-width* *robot-window-height*
    *terrain-buffer-array* 2 2 0 0)))



efun draw-to (a-point a-window)
global variables : *start-point*
(tv:sheet-force-access (a-window)
  (send a-window ':draw-line (first *start-point*)
    (- *max-y* (second *start-point*))
    (first a-point)
    (- *max-y* (second a-point))   tv:alu-ior))
(setq *start-point* a-point))



efun draw-to-d (a-point a-window)
global variables : *start-point*
(tv:sheet-force-access (a-window)
  (send a-window ':draw-fat-line (first *start-point*)
    (- *max-y* (second *start-point*))
    (first a-point)
    (- *max-y* (second a-point))   tv:alu-ior))
(setq *start-point* a-point))



efun erase-to (a-point a-window)
global variables : *start-point*
(tv:sheet-force-access (a-window)
  (send a-window ':draw-line (first *start-point*)
    (- *max-y* (second *start-point*))
    (first a-point)
```

```
      (- *max-y* (second a-point))   tv:alu-andca))
  (setq *start-point* a-point))



(defun get-keyboard-input()
; This is not for the graphics, but this function uses Zeta LISP.
; This is the reason why this function is in Zeta graphic package.
  (send terminal-io :tyi-no-hang))



(defun make-robot-window ()
  (setq *robot-display-window* (tv:make-window 'tv:window
                            ':blinker-p nil
                            ':edges-from :mouse
                            ':borders 2
                            ':label "robot-display-window"
                            ':name "robot-display-window"
                                ':save-bits t
                            ':expose-p t))
  (let* ((r-w (send *robot-display-window* :width))
     (r-h (send *robot-display-window* :height))
     (r-x nil) (r-y nil))
    (multiple-value (r-x r-y) (send *robot-display-window* :position))
    (setq *robot-window* (tv:make-window 'tv:window
                    ':position (list r-x r-y)
                    ':width r-w
                    ':height r-h
                    ':blinker-p nil
                    ':borders 2
                    ':label "robot-window"
                    ':name "robot-window"
                    ':save-bits t
                    ':expose-p nil))
    (setq *terrain-buffer* (tv:make-window 'tv:window
                      ':position (list r-x r-y)
                      ':width r-w
                      ':height r-h
                      ':blinker-p nil
                      ':borders 2
                      ':label "terrain-buffer"
                      ':name "terrain-buffer"
                      ':save-bits t
                      ':expose-p nil))
    (setq *max-y* (send *robot-window* :inside-height)))
  (setq *robot-display-window-array* (send *robot-display-window* :bit-array))
  (setq *robot-window-array* (send *robot-window* :bit-array))
  (setq *robot-window-width* (send *robot-window* :inside-width))
  (setq *robot-window-height* (send *robot-window* :inside-height))
  (setq *terrain-buffer-array* (send *terrain-buffer* :bit-array)))



(defun make-visible ()
  (send *robot-display-window* :bitblt
    tv:alu-seta *robot-window-width* *robot-window-height*
    *robot-window-array* 2 2 0 0))



(defun move-to (a-point)
; global variables : *start-point*
; This function just changes *start-point*.
```

```
(setq *start-point* a-point))


(defun save-terrain-to-terrain-buffer()
(tv:sheet-force-access (*terrain-buffer*)
  (send *terrain-buffer* :bitblt
    tv:alu-seta *robot-window-width* *robot-window-height*
    *robot-window-array* 2 2 0 0)))
```

```
; -*- Mode: LISP; Syntax: Common-lisp; Package: BODY -*-

efflavor H-calculator((sampling-time 0.1) H
                      old-H)
        ()
:initable-instance-variables)


efmethod (init H-calculator)
        ()
library fucntion : ident
(setf H  (ident))
(setf (aref H 0 3) 6.5)
(setf (aref H 1 3) 19.5)
(setf (aref H 2 3) 5.4)
H)


efmethod (new-H H-calculator)
        (body-trans-rate body-rotate-rate)
(setf H
      (orthogonalization
        (get-new-H
         H
         (get-del-H
           H
           (get-delta body-trans-rate body-rotate-rate sampling-time))))))



efmethod (save H-calculator)
        ()
(setf old-H H))


efmethod (restore H-calculator)
        ()
(setf H old-H))



**************************************************"

 H-calculator.new-H

**************************************************"


efun get-delta (body-trans-rate body-rotate-rate sampling-time)
(let* ((del-trans-x  (* (first  body-trans-rate)  sampling-time))
       (del-trans-y  (* (second body-trans-rate)  sampling-time))
       (del-trans-z  (* (third  body-trans-rate)  sampling-time))
       (del-rotate-x (* (first  body-rotate-rate) sampling-time))
       (del-rotate-y (* (second body-rotate-rate) sampling-time))
       (del-rotate-z (* (third  body-rotate-rate) sampling-time)))
  (list (list del-trans-x  del-trans-y  del-trans-z)
        (list del-rotate-x del-rotate-y del-rotate-z))))



efun get-del-H (H delta-trans-rotate)
```

```
; math lib : ident
  (let* ((H-del (ident))        ; initialze identity matirix
         (delta-trans  (first  delta-trans-rotate))
         (delta-rotate (second delta-trans-rotate)))
    (setf (aref H-del 0 0) 0)
    (setf (aref H-del 1 0) (third delta-rotate))
    (setf (aref H-del 2 0) (- (second delta-rotate)))
    (setf (aref H-del 0 1) (- (third delta-rotate)))
    (setf (aref H-del 1 1) 0)
    (setf (aref H-del 2 1) (first delta-rotate))
    (setf (aref H-del 0 2) (second delta-rotate))
    (setf (aref H-del 1 2) (- (first delta-rotate)))
    (setf (aref H-del 2 2) 0)
    (setf (aref H-del 0 3) (first  delta-trans))
    (setf (aref H-del 1 3) (second delta-trans))
    (setf (aref H-del 2 3) (third  delta-trans))
    (setf (aref H-del 3 3) 0)
    (matrixmult H H-del)))


(defun get-new-H (H del-H)
  (matrixadd H del-H))
```

```
 -*- Package: LEG; Mode: LISP; Syntax: Common-lisp; Base: 10 -*-

************************************************************

leg flavor definition

************************************************************


efflavor leg (name owner plan-machine control-machine
                   executor contact-sensor tkm-calculator
                   foothold-finder exchanged-leg
                   foothold tkm foothold-list tkm-list tkm-p
                   reserved-foothold reserved-tkm)
        ()
initable-instance-variables
readable-instance-variables)


fmethod (init leg)
        (H)
 (setf contact-sensor  (make-instance 'contact-sensor  :owner self))
 (setf executor        (make-instance 'executor        :owner self))
 (setf control-machine (make-instance 'control-state-machine :owner self))
 (setf plan-machine    (make-instance 'plan-state-machine    :owner self))
 (setf tkm-calculator  (make-instance 'tkm-calculator  :owner self))
 (setf foothold-finder (make-instance 'foothold-finder :owner self))
 (setf foothold (init executor name H))
 (init contact-sensor name)
 (init control-machine name)
 (init plan-machine name)
 (init tkm-calculator name)
 (init foothold-finder name))


fmethod (contact-confirm leg)
        ()
contact-p contact-sensor))


fmethod (do-planned-motion leg)
        ()
change plan-machine)
change control-machine)
move executor (get-H1 owner) (get-inv-H1 owner)
     (get-body-trans-rate1 owner)
     (get-body-rotate-rate1 owner))
sensing contact-sensor))


fmethod (get-H1 leg)
        ()
get-H1 owner))


fmethod (interlock-confirm leg)
        ()
ay add stable-without-p self
if (contact-confirm exchanged-leg)
```

```
      t
      nil))


(defmethod (leg-pos-wrt-body leg)
           ()
  (leg-pos-wrt-body executor))


(defmethod (lift-able leg)
           ()
  (if (equal (state-name plan-machine) 'eligible-to-lift)
      self
      nil))


(defmethod (lift-ok leg)
           ()
  (lift-ok owner name))


(defmethod (lifted leg)
           ()
  (lifted owner name))


(defmethod (new-foothold leg)
           ()
  (cond ((car foothold-list)
         (set-max self)
         t)
        (t
         nil)))


(defmethod (permitted-cell leg)
           (t-cell)
  (permitted-cell owner t-cell))


(defmethod (place-able leg)
           ()
  (if (equal (state-name plan-machine) 'available-leg)
      self
      nil))


(defmethod (projected-pos leg)
           ()
  (get-desired-pos executor))


(defmethod (select-foothold leg)
           ()
; out-list: ((max-foothold max-tkm) (foothold-list) (tkm-list))
  (let* ((H (get-H6 owner))
         (inv-H (get-inv-H6 owner))
```

```
      (body-trans-rate (get-body-trans-rate10 owner))
      (body-rotate-rate (get-body-rotate-rate10 owner))
      (estimated-support-plane
        (get-estimated-support-plane owner))
      (out-list
        (find-foothold foothold-finder
                       H inv-H body-trans-rate body-rotate-rate
                       estimated-support-plane)))
  (setf foothold (first (first out-list)))
  (setf reserved-foothold foothold)
  (setf tkm (second (first out-list)))
  (setf reserved-tkm tkm)
  (setf foothold-list (second out-list))
  (setf tkm-list (third out-list))))


efmethod (send-decision leg)
         (a-decision)
(send-decision plan-machine a-decision))


efmethod (send-decision leg :after)
         (a-decision)
(if (equal a-decision 'place)
    (set-desired-pos executor foothold)))


efmethod (send-exchange leg)
         (a-leg)
(setf exchanged-leg a-leg))


efmethod (set-max leg)
         ()
(do ((footholds (cdr foothold-list) (cdr footholds))
     (tkms (cdr tkm-list) (cdr tkms))
     (max-foothold (car foothold-list))
     (max-tkm (car tkm-list))
     (out-footholds) (out-tkms))
    ((null footholds)
     (setf foothold max-foothold)
     (setf tkm max-tkm)
     (setf foothold-list out-footholds)
     (setf tkm-list out-tkms))
  (cond ((> (car tkms) max-tkm)
         (setf max-foothold (car footholds))
         (setf max-tkm (car tkms)))
        (t
         (setf out-footholds
               (cons (car footholds) out-footholds))
         (setf out-tkms
               (cons (car tkms) out-tkms))))))


efmethod (stable-without-p leg)
         ()
(stable-without-p owner self))
```

```
(defmethod (supporting leg)
           ()
  (cond ((equal (state-name plan-machine) 'planned-contact)
         self)
        ((equal (state-name plan-machine) 'eligible-to-lift)
         self)
        (t nil))
  )


(defmethod (supporting-p leg)
           ()
  (cond ((equal (state-name control-machine) 'contact)
         self)
        ((equal (state-name control-machine) 'support)
         self)
        (t nil))
  )


(defmethod (terrain-point leg)
           (t-cell)
  (terrain-point owner t-cell))



(defmethod (TKM-limit leg)
           ()
  (cond ((null tkm)
         self)
        ((< tkm 0.1)
         self)
        (t
         nil)))


(defmethod (TKM-limit-p leg)
           ()
  (cond ((null tkm-p)
         self)
        ((< tkm-p 0.5)
         self)
        (t nil)))


(defmethod (update-tkm leg)
           ()
  (let ((body-trans-rate (get-body-trans-rate10 owner))
        (body-rotate-rate (get-body-rotate-rate10 owner))
        (inv-H (get-inv-H10 owner)))
    (setf tkm (find-tkm tkm-calculator
                        (to-body-transform inv-H foothold)
                        body-trans-rate body-rotate-rate)))
    )


(defmethod (update-tkm-p leg)
```

```
        ()
(let ((body-trans-rate-p (get-body-trans-rate1 owner))
      (body-rotate-rate-p (get-body-rotate-rate1 owner))
      (inv-H-p (get-inv-H1 owner)))
  (setf tkm-p (find-tkm tkm-calculator
                        (to-body-transform inv-H-p foothold)
                        body-trans-rate-p body-rotate-rate-p)))
)


efmethod (with-foothold leg)
        ()
(cond (reserved-foothold
       (setf foothold reserved-foothold)
       (setf tkm reserved-tkm)
       self)
      (t nil)))
```

```
; -*- Mode: LISP; Package: USER; Syntax: Common-lisp -*-
efun load-files()

(load "object:packdef")

(load "object:math")

(load "object:graph")
(load "object:display")

(load "object:vision")

(load "object:tkm")
(load "object:foothold")
(load "object:sensor")
(load "object:executor")
(load "object:control-machine")
(load "object:plan-machine")
(load "object:leg")

(load "object:stability")
(load "object:support-plane")
(load "object:h-calculator")
(load "object:command-regulator")
(load "object:terrain-regulator")
(load "object:body-controller")
(load "object:body")

(load "object:robot")

(cp:execute-command "set package" "robot-rules")
(load "object:robot4"))

oad-files)
```

```
 -*- Mode: LISP; Package: ROBOT-MATH; Syntax: Common-lisp -*-


**************************************************"

 robot math library

**************************************************"


efun arc-cos (s)
(acos s))


efun atan2 (y x)
(if (> (abs x) 0.000001)    ; not zero
    (if (> x 0)
        (atan (/ y x))
        (+ (atan (/ y x)) (* (signum y) PI)))
    (* (signum y) (/ PI 2))))


efun col-mul(mat col1 col2)
let ((sum 0))
 (dotimes (i 4)
    (setf sum (+ sum (* (aref mat i col1) (aref mat i col2)))))
um))


efun counting(a-list)
do ((a-list a-list (cdr a-list))
    (i 0 (+ i 1)))
   ((null a-list) i)))


efun crossprod (vect1 vect2)
let* ((x1 (first  vect1)) (x2 (first  vect2))
      (y1 (second vect1)) (y2 (second vect2))
      (z1 (third  vect1)) (z2 (third  vect2))
      (x (- (* y1 z2) (* y2 z1)))
      (y (- (* x2 z1) (* x1 z2)))
      (z (- (* x1 y2) (* x2 y1))))
 (list x y z)))


efun delete-list (a-list b-list)   ; delete a-list from b-list
do  ((deleting-list a-list (cdr deleting-list))
     (deleted-list b-list))
   ((null deleting-list) deleted-list)
 (setf deleted-list (remove (car deleting-list)
                            deleted-list :test 'equal))))


efmacro dequeue (queue)
(prog1 (car ,queue)
       (setf ,queue (cdr ,queue))))
```

```
(defun dotprod (vect1 vect2)
; No dimension limitation !!!
  (apply '+ (mapcar '* vect1 vect2)))


(defmacro enqueue (queue-name element)
; globals : queue-name
; Value of recover field of command is a list.
; Two recover command is possible for one sampling-time.
; structure of QUEUE :  (first second third ... last)
  `(setq ,queue-name (nconc ,queue-name (list ,element))))


(defmacro empty-queue (queue)
  `(setq ,queue '()))


(defun ident()
  (make-array '(4 4):initial-contents
              '((1 0 0 0)
                (0 1 0 0)
                (0 0 1 0)
                (0 0 0 1))))


(defun magnitude (a-vector)
  (sqrt (dotprod a-vector a-vector)))


(defun magvect (const vect)
; magvect = const * vect
  (mapcar (lambda (a-element)
            (* const a-element))
          vect))


(defun matrixadd (mt1 mt2)
  (let ((mt3 (ident)))
    (dotimes (i 4)
      (dotimes (j 4)
        (setf (aref mt3 i j) (+ (aref mt1 i j) (aref mt2 i j)))))
    mt3))


(defun matrixinv(mat)
  (let ((px  (- (col-mul mat 0 3)))
        (py  (- (col-mul mat 1 3)))
        (pz  (- (col-mul mat 2 3)))
        (matrix (transpose mat)))
    (setf (aref matrix 3 0) 0)  (setf (aref matrix 3 1) 0)
    (setf (aref matrix 3 2) 0)  (setf (aref matrix 3 3) 1)
    (setf (aref matrix 0 3) px) (setf (aref matrix 1 3) py)
    (setf (aref matrix 2 3) pz)
    matrix))
```

```
efun matrixmult (mt1 mt2)
(let ((mat (make-array '(4 4)))) ;it defines 0 through 3. (4 is not included)
  (dotimes (i 4)      ; will repeat i=0, 1, 2, and 3. (not 4)
    (dotimes (j 4)
      (setf (aref mat i j) 0)          ; initialize to zero
      (dotimes (k 4)
        (setf (aref mat i j) (+ (aref mat i j) (* (aref mt1 i k)
                                                  (aref mt2 k j)))))))
  mat))


efun nil-list(a-list)
(do ((a-list a-list (cdr a-list))
     (not-nil nil))
    ((null a-list) (not not-nil))
  (if (car a-list)
      (setf not-nil t))))


efun normalize-vector (a-vector)
(let ((m (magnitude a-vector)))
  (if (< m 0.0000001)
      (list 0 0 0)
      (magvect (/ 1.0 m) a-vector))))


efun orthogonalization (mt)
              ; Gram-Schimit orthogonalization process
(let* ((mx (ident))
       (tx (aref mt 0 3)) (ty (aref mt 1 3)) (tz (aref mt 2 3))

       (x1 (aref mt 0 0)) (x2 (aref mt 0 1)) (x3 (aref mt 0 2))
       (y1 (aref mt 1 0)) (y2 (aref mt 1 1)) (y3 (aref mt 1 2))
       (z1 (aref mt 2 0)) (z2 (aref mt 2 1)) (z3 (aref mt 2 2))
       (m1 (magnitude (list x1 y1 z1)))
       (x1 (/ x1 m1))
       (y1 (/ y1 m1))
       (z1 (/ z1 m1))
       (a  (dotprod (list x1 y1 z1) (list x2 y2 z2)))
       (x2 (- x2 (* a x1)))
       (y2 (- y2 (* a y1)))
       (z2 (- z2 (* a z1)))
       (m2 (magnitude (list x2 y2 z2)))
       (x2 (/ x2 m2))
       (y2 (/ y2 m2))
       (z2 (/ z2 m2)))
  (setf (aref mx 0 0) x1) (setf (aref mx 0 1) x2) (setf (aref mx 0 2) x3)
  (setf (aref mx 1 0) y1) (setf (aref mx 1 1) y2) (setf (aref mx 1 2) y3)
  (setf (aref mx 2 0) z1) (setf (aref mx 2 1) z2) (setf (aref mx 2 2) z3)
  (setf (aref mx 0 3) tx) (setf (aref mx 1 3) ty) (setf (aref mx 2 3) tz)
  mx))


efun plane-transform ( plane matrix )
Transformed-Plane = Plane * Matrix
plane is defined as ((a b c) d). (a b c) is unit normal. d is -(distance).
```

```
  (let* ((new-a nil)
         (new-b nil)
         (new-c nil)
         (new-d nil)
         (old-unit-normal (car plane))
         (old-d (cadr plane))
         (old-a (first  old-unit-normal))
         (old-b (second old-unit-normal))
         (old-c (third  old-unit-normal))
         (mag nil))
    (setf new-a (+ (* old-a (aref matrix 0 0)) (* old-b (aref matrix 1 0))
                   (* old-c (aref matrix 2 0))))
    (setf new-b (+ (* old-a (aref matrix 0 1)) (* old-b (aref matrix 1 1))
                   (* old-c (aref matrix 2 1))))
    (setf new-c (+ (* old-a (aref matrix 0 2)) (* old-b (aref matrix 1 2))
                   (* old-c (aref matrix 2 2))))
    (setf new-d (+ (* old-a (aref matrix 0 3)) (* old-b (aref matrix 1 3))
                   (* old-c (aref matrix 2 3)) old-d))
    (setf mag (magnitude (list new-a new-b new-c)))
    (if (< (abs mag) 0.0000001)
        (print "Error in PlaneTransform")
        (list (list (/ new-a mag) (/ new-b mag) (/ new-c mag))
              (/ new-d mag)))))




(defun plane-distance (plane velocity position)
; Plane (X - Q)N = 0 ,  straight line X = P + tA.
; t = ( Q - P )N / ( AN )  if A is normalized then t is signed distance.
; if t is infinitive then plane-distance  returnes nil.
; plane-distance returns t.
  (let* ((A (normalize-vector velocity))
         (N (first plane))
         (dis (- (second plane)))
         (Q   (magvect dis N)) ; magvect = const * vector
         (P   position)
         (Q_P (vectsub Q P))
         (AN  (dotprod A N))
         (numerator (dotprod Q_P N)))
    (if (< (abs AN) 0.0000001)  ; no crossing
        nil                     ; returns nil
        (/ numerator AN))))



(defun plane-intersection (a-line a-plane)
; a-line  ((direction) (point))  X = P + tA.
; a-plane ((unit-normal) -dist)  (X - Q)N = 0.
  (let* ((velocity (normalize-vector (first a-line)))
         (position (second a-line))
         (t-value (plane-distance a-plane velocity position)))
    (if t-value
        (vectadd position (magvect t-value velocity))
        nil))) ; no intersection



(defun plane-normal-distance (a-plane a-point)
; vector-type-plane (a b c d)
; paul-type-point   transpose(x y z 1)
```

```
(let* ((unit-normal (first  a-plane))
       (dis         (second a-plane))
       (vector-type-plane (reverse (cons dis (reverse unit-normal))))
       (paul-type-point   (reverse (cons 1   (reverse a-point)))))
  (dotprod vector-type-plane paul-type-point)))



efun rotatemat(axis angle) ; array index starts from 0 not 1.
return rotatematrix   angle :radian    axis : x y or z
(let ((mat (ident))
      (cosa (cos angle))
      (sina (sin angle)))
  (case axis
        (x-axis
          (setf (aref mat 1 1)  cosa)      (setf (aref mat 1 2) (- sina))
          (setf (aref mat 2 1)  sina)      (setf (aref mat 2 2)  cosa))
        (y-axis
          (setf (aref mat 0 0)  cosa)      (setf (aref mat 0 2)  sina)
          (setf (aref mat 2 0) (- sina))   (setf (aref mat 2 2)  cosa))
        (z-axis
          (setf (aref mat 0 0)  cosa)      (setf (aref mat 0 1) (- sina))
          (setf (aref mat 1 0)  sina)   (setf (aref mat 1 1)  cosa)))
  mat))  ; returns this value.



efun to-body-transform (inv-H points-wrt-earth)
returns points-wrt-body
(if (listp (first points-wrt-earth))              ; test multi-points
    (do ((points points-wrt-earth (cdr points)) ; multi-points case
         (out-points nil))
        ((null points) (reverse out-points))
      (setf out-points (cons (transform inv-H (car points)) out-points)))
    (transform inv-H points-wrt-earth)))         ; single point case



efun to-earth-transform (H points-wrt-body)
returns points-wrt-earth
(if (listp (first points-wrt-body))              ; test multi-points
    (do ((points points-wrt-body (cdr points))  ; multi-points case
         (out-points nil))
        ((null points) (reverse out-points))
      (setf out-points (cons (transform H (car points)) out-points)))
    (transform H points-wrt-body)))              ; single point case



efun transform(mat point)  ; array index starts from 0 not 1.
(let ((x (car point))
      (y (cadr point))
      (z (if (caddr point) (caddr point) 0)))
  (list (+ (* x (aref mat 0 0)) (* y (aref mat 0 1)) (* z (aref mat 0 2))
           (aref mat 0 3))
        (+ (* x (aref mat 1 0)) (* y (aref mat 1 1)) (* z (aref mat 1 2))
           (aref mat 1 3))
        (+ (* x (aref mat 2 0)) (* y (aref mat 2 1)) (* z (aref mat 2 2))
           (aref mat 2 3)))))



efun transmat (x y z)
```

```
; returns translational marix
  (let ((matrix (ident)))
    (setf (aref matrix 0 3) x)
    (setf (aref matrix 1 3) y)
    (setf (aref matrix 2 3) z)
    matrix))



(defun transpose (mat)
  (let ((matrix (make-array '(4 4))))
    (dotimes (i 4)
      (dotimes (j 4)
        (setf (aref matrix i j) (aref mat j i))))
    matrix))



(defun unit-crossprod (vect1 vect2)
; generate unitnormal vector of vect1 X vect2
  (let* ((x1 (first  vect1)) (x2 (first  vect2))
         (y1 (second vect1)) (y2 (second vect2))
         (z1 (third  vect1)) (z2 (third  vect2))
         (x (- (* y1 z2) (* y2 z1)))
         (y (- (* x2 z1) (* x1 z2)))
         (z (- (* x1 y2) (* x2 y1)))
         (m (sqrt (+ (* x x) (* y y) (* z z)))))
    (list (/ x m) (/ y m) (/ z m))))



(defun vectadd (vect1 vect2)
; vectsub = vect1 + vect2
; no limit in dimension
  (mapcar '+ vect1 vect2))



(defun vectsub (vect1 vect2)
; vectsub = vect1 - vect2
; no limit in dimension
  (mapcar '- vect1 vect2))
```

```
; -*- Mode: LISP; Syntax: Common-lisp; Package: USER -*-
efpackage robot-common
(:use)
(:export
  leg1 leg2 leg3 leg4 leg5 leg6
  lift place exchange))


efpackage robot-math
(:use symbolics-common-lisp)
(:export
  vectsub
  vectadd
  unit-crossprod
  transpose
  transmat
  transform
  to-earth-transform
  to-body-transform
  rotatemat
  plane-normal-distance
  plane-intersection
  plane-distance
  plane-transform
  orthogonalization
  normalize-vector
  nil-list
  matrixmult
  matrixinv
  matrixadd
  magvect
  magnitude
  ident
  empty-queue
  enqueue
  dotprod
  dequeue
  delete-list
  crossprod
  counting
  col-mul
  atan2
  arc-cos
  x-axis y-axis z-axis))


lefpackage terrain
(:use robot-math symbolics-common-lisp)
(:export
  init-terrain
  graph-terrain
    display modify
    create
    permitted-cell get-height in-side-of-whole-terrain terrain-point
  graph-asv
    init-data
    display
  joystick
    reset
    get-joy-value))


lefpackage robot
```

```
  (:use robot-common robot-math  symbolics-common-lisp pl-user))


(defpackage leg
  (:use robot-common robot-math terrain robot symbolics-common-lisp)
  (:export
    leg
      init
      leg-name
      leg-foothold
      leg-tkm
      send-exchange
      do-planned-motion
      simulation-output
      leg-pos-wrt-body
      lift-able
      place-able
      supporting
      supporting-p
      send-decision
      select-foothold
      update-tkm
      update-tkm-p
      new-foothold
      tkm-limit
      with-foothold
      tkm-limit-p
      ))


(defpackage body
  (:use robot-common robot-math leg  symbolics-common-lisp)
  (:export
    body
      init
      get-H1
      get-inv-H1
      get-H6
      get-inv-H6
      get-inv-H10
      get-body-trans-rate1
      get-body-rotate-rate1
      get-body-trans-rate10
      get-body-rotate-rate10
      get-estimated-support-plane
      stable
      calculate-motion
      more-stable
      slow-down
      speed-up
      near-dead-lock-test
      stable-p
      stable-p-m
      stable-m
      modify-command
      modify-command-p
      restore-command
      restore-command-p
      stop-motion
      restore-motion
   ))
```

```
efpackage vision
(:use robot-common robot-math leg robot terrain symbolics-common-lisp)
(:export
  vision-system
     init
     scanning
     permitted-cell
     terrain-point))


efpackage robot
(:use robot-common robot-math terrain leg body vision symbolics-common-lisp pl-user)
(:export
  create-terrain
  kill-terrain
  inits
  graphical_display
  execute_planned_motion
  send_decision
  stable_p
  max_sm_leg
  stable_p_m
  modify_command
  stable_without
  smallest_tkm_leg
  update_robot_status
  read_joystick
  restore_command
  slow_down_robot
  speed_up_robot
  leg_with_new_foothold
  at_tkm_limit
  do_recovery
  has_more_tkm
    get-H1
    get-inv-H1
    get-H6
    get-inv-H6
    get-inv-H10
    get-body-trans-rate1
    get-body-rotate-rate1
    get-body-trans-rate10
    get-body-rotate-rate10
    get-estimated-support-plane
    lift-ok
    lifted
    stable-without-p
    terrain-point
    ))


lefpackage robot-rules
(:use prolog-global robot-common robot symbolics-common-lisp ))
```

```
s:set-logical-pathname-host "object"
 :translations '(("object:**;*.*.*" "sym4:>kwak>object-wp>*.*.*")))
```

```
; -*- Mode: LISP; Syntax: Common-lisp; Package: LEG -*-

;*******************************************************

 plan-state flavor definition

;*******************************************************


defflavor plan-state ((decision nil) (observation nil) (command nil)
                      (condition nil))
          (state)
 :initable-instance-variables)


defmethod (generate-command plan-state)
          ()
 command)


defmethod (change plan-state)
          (given-decision observed-state given-condition)
 (cond ((and decision (listp decision))
        (cond ((equal given-decision (first decision))
               (first next-state))
              ((equal given-decision (second decision))
               (second next-state))
              (t self)))
       (condition
        (if (and (equal given-condition condition)
                 (equal observed-state observation))
            next-state
            self))
       (t
        (cond ((equal observed-state observation)
               next-state)
              ((equal given-decision decision)
               next-state)
              (t self)))))



;*******************************************************

  plan-state-machinie flavor definition

;*******************************************************


defflavor plan-state-machine ((decision nil) (observation nil)
                              (condition nil) (lift-ready-flag nil)
                              control-machine)
          (state-machine)
 :initable-instance-variables)


defmethod (init plan-state-machine)
          (leg-name)
 (if (member leg-name '(leg1 leg4 leg5))
     (init-plan-machine self 'eligible-to-lift)
```

```
      (init-plan-machine self 'available-leg))
  (setf control-machine (leg-control-machine owner)))


(defmethod (init-plan-machine plan-state-machine)
           (a-state-name)
  (let (available-leg planned-contact eligible-to-lift
        planned-lift  actual-lift planned-exchange)
    (setf actual-lift
          (make-instance 'plan-state
                         :name 'actual-lift
                         :observation 'ready
                         :command 'recover-command))
    (setf planned-lift
          (make-instance 'plan-state
                         :name 'planned-lift :condition 'stable-without
                         :observation 'support
                         :next-state actual-lift))
    (setf planned-exchange
          (make-instance 'plan-state
                         :name 'planned-exchange :condition 'interlock-confirm
                         :observation 'support
                         :next-state actual-lift))
    (setf eligible-to-lift
          (make-instance 'plan-state
                         :name 'eligible-to-lift
                         :decision '(lift exchange)
                         :next-state (list planned-lift planned-exchange)))
    (setf planned-contact
          (make-instance 'plan-state
                         :name 'planned-contact :observation 'contact
                         :command 'deploy-command
                         :next-state eligible-to-lift))
    (setf available-leg
          (make-instance 'plan-state
                         :name 'available-leg :decision 'place
                         :next-state planned-contact))
    (set-next-state actual-lift available-leg)

    (setf state (cond ((equal a-state-name (state-name available-leg))
                       available-leg)
                      ((equal a-state-name (state-name planned-contact))
                       planned-contact)
                      ((equal a-state-name (state-name eligible-to-lift))
                       eligible-to-lift)
                      ((equal a-state-name (state-name planned-lift))
                       planned-lift)
                      ((equal a-state-name (state-name planned-exchange))
                       planned-exchange)
                      ((equal a-state-name (state-name actual-lift))
                       actual-lift)))
    )
  )


(defmethod (change plan-state-machine :before)
           ()
  (setf observation (state-name control-machine))
  (cond ((and (equal (state-name state) 'planned-exchange)
              (interlock-confirm owner)
              (stable-without-p owner)
              (lift-ok owner))
```

```
       (setf lift-ready-flag t)
       (setf condition 'interlock-confirm))
      ((and (equal (state-name state) 'planned-lift)
            (stable-without-p owner)
            (lift-ok owner))
       (setf lift-ready-flag t)
       (setf condition 'stable-without))
      (t
       (setf condition nil)
       (setf lift-ready-flag nil)))
)


efmethod (change plan-state-machine)
         ()
(setf state (change state decision observation condition)))


efmethod (change plan-state-machine :after)
         ()
(send-command control-machine
              (generate-command state))
(if (and lift-ready-flag
         (equal (state-name self) 'actual-lift))
    (lifted owner)))


efmethod (send-decision plan-state-machine)
         (a-decision)
(setf decision a-decision))
```

```
; -*- Mode: LISP; Package: ROBOT; Syntax: Common-lisp; Base: 10 -*-

lefvar asv)

;*******************************************************

  robot flavor definition

;*******************************************************

lefflavor robot(legs body vision-system joystick
                    (lift-able-legs nil)
                    (place-able-legs nil) (supporting-legs nil)
                    (supporting-p-legs nil)
                    (joy-command '(0 0 0)) lift-queue lift-flag)
         ()
:initable-instance-variables)


etf asv (make-instance 'robot))


lefmethod (init robot)
         ()
(init-data graph-asv)
(empty-queue lift-queue)
(setf lift-flag t)
(setf joystick (make-instance 'joystick))
(reset joystick)
(setf vision-system (make-instance 'vision-system :owner self))
(init vision-system)
(let ((H))
  (setf body (make-instance 'body :owner self))
  (setf H (init body))
  (setf legs (list
               (make-instance 'leg :name 'leg1  :owner self)
               (make-instance 'leg :name 'leg2  :owner self)
               (make-instance 'leg :name 'leg3  :owner self)
               (make-instance 'leg :name 'leg4  :owner self)
               (make-instance 'leg :name 'leg5  :owner self)
               (make-instance 'leg :name 'leg6  :owner self)
              ))
  (mapcar (lambda (a-leg) (init a-leg H)) legs))
)


defmethod (find-lift-able-legs robot)
         ()
(delete nil (mapcar 'lift-able legs)))


defmethod (find-place-able-legs robot)
         ()
(delete nil (mapcar 'place-able legs)))


defmethod (find-supporting-legs robot)
         ()
```

```
  (delete nil (mapcar 'supporting legs)))


(defmethod (find-supporting-p-legs robot)
           ()
  (delete nil (mapcar 'supporting-p legs)))


(defmethod (get-body-rotate-rate1 robot)
           ()
  (get-body-rotate-rate1  body))


(defmethod (get-body-rotate-rate10 robot)
           ()
  (get-body-rotate-rate10  body))


(defmethod (get-body-trans-rate1 robot)
           ()
  (get-body-trans-rate1 body))


(defmethod (get-body-trans-rate10 robot)
           ()
  (get-body-trans-rate10 body))


(defmethod (get-estimated-support-plane robot)
           ()
  (get-estimated-support-plane body))


(defmethod (get-H1 robot)
           ()
  (get-H1 body))


(defmethod (get-H6 robot)
           ()
  (get-H6 body))


(defmethod (get-inv-H1 robot)
           ()
  (get-inv-H1 body))


(defmethod (get-inv-H6 robot)
           ()
  (get-inv-H6 body))
```

```
defmethod (get-inv-H10 robot)
          ()
 (get-inv-H10 body))


defmethod (lift-ok robot)
          (leg-name)
 (cond (lift-flag
        (cond ((equal leg-name (leg-name (first lift-queue)))
               (setf lift-flag nil)
               t)
              (t
               nil)))
       (t nil)))


defmethod (lifted robot)
          (leg-name)
 (if (equal leg-name (leg-name (first lift-queue)))
     (dequeue lift-queue)
     (print (list "error in lifting" leg-name))))


defmethod (permitted-cell robot)
          (t-cell)
 (permitted-cell vision-system t-cell))


defmethod (scanning robot)
          ()
 (scanning vision-system))


defmethod (stable-without-p robot)
          (a-leg)
 (stable-p body
           (remove a-leg supporting-p-legs)))


defmethod (terrain-point robot)
          (t-cell)
 (terrain-point vision-system t-cell))




**********************************************************

 terrain interface functions

**********************************************************


defun create-terrain()
 (init-terrain))
```

```
(defun kill-terrain()
  (zl-user:kill-all-windows))



;*********************************************************
;
; prolog interface robot methods
;
;*********************************************************



(defmethod (at-tkm-limit robot)
           ()
  (let ((limit-leg
          (car (delete nil
                (mapcar 'TKM-limit lift-able-legs)))))
    (setf supporting-legs (remove limit-leg
                                  supporting-legs))
    (setf lift-able-legs (remove limit-leg
                                 lift-able-legs))
    limit-leg))



(defmethod (check-stability-p robot)
           ()
  (stable-p-m body supporting-p-legs (first lift-queue)))



(defmethod (check-tkm-limit-p robot)
           ()
  (delete nil
             (mapcar 'TKM-limit-p supporting-p-legs)))



(defmethod (do-recovery robot)
           ()
  (car
    (delete nil
            (mapcar 'with-foothold place-able-legs))))



(defmethod (execute-planned-motion robot)
           ()
  (mapcar 'do-planned-motion legs))



(defmethod (graphical-display robot)
           ()
  (display graph-asv (get-H1 body)
           (mapcar 'leg-pos-wrt-body legs))
  )
```

```
defmethod (has-more-tkm robot)
          (leg1 leg2)
 (> (leg-tkm leg1)
    (leg-tkm leg2)))



defmethod (leg-with-new-foothold robot)
          ()
 return a-leg with new-foothold.
 (do ((new-foothold-flags (mapcar 'new-foothold place-able-legs)
                          (mapcar 'new-foothold place-able-legs))
      (a-leg nil))
     ((or (nil-list new-foothold-flags)
          a-leg)
      (if a-leg a-leg nil))
   (setf a-leg (max-sm-leg self nil))))



defmethod (max-sm-leg robot)
          (a-leg)
 (if place-able-legs
     (do ((legs (cdr place-able-legs) (cdr legs))
          (largest-leg (car place-able-legs) largest-leg)
          (temp-support-legs (remove a-leg supporting-legs)))
         ((null legs)
          (if (and
                (leg-foothold largest-leg)
                (stable body (cons largest-leg temp-support-legs)))
              largest-leg
              nil))
       (if (more-stable body temp-support-legs
                        (car legs) largest-leg)
           (setf largest-leg  (car legs))))
     nil))



defmethod (modify-command robot)
          ()
 (modify-command body))



defmethod (wait-for-lift robot)
          ()
 (delete nil
         (mapcar 'lift-not-done supporting-p-legs)))


defmethod (read-joystick robot)
          ()
 (let ((joy-value (get-joy-value joystick)))
   (if (fourth joy-value)
       (modify graph-terrain))
   (setf joy-command
         (reverse (cdr (reverse joy-value))))))



defmethod (restore-command robot)
          ()
```

```
  (restore-command body))


(defmethod (send-decision robot)
           (leg1 leg2 a-decision)
  (cond ((equal a-decision 'exchange)
         (enqueue lift-queue leg1))
        ((equal a-decision 'lift)
         (enqueue lift-queue leg1)))
  (cond ((equal a-decision 'exchange)
         (send-decision leg1 a-decision)
         (send-decision leg2 'place)
         (send-exchange leg1 leg2))
        (t
         (send-decision leg1 a-decision))))


(defmethod (smallest-tkm-leg robot)
           ()
; select smallest-TKM-leg
; tkm is nil or positive
  (do ((legs (cdr lift-able-legs) (cdr legs))
       (smallest-leg  (car lift-able-legs))
       (smallest-tkm nil) (tkm nil))
      ((null legs) smallest-leg)
    (setf smallest-tkm (if (leg-tkm smallest-leg)
                           (leg-tkm smallest-leg) -1000))
    (setf tkm (if (leg-tkm (car legs))
                  (leg-tkm (car legs)) -1000))
    (if (> smallest-tkm tkm) (setf smallest-leg (car legs)))
    (if (and (equal smallest-tkm -1000) (equal tkm -1000))
        "Error : more than one legs are out of kinematic limit")))


(defmethod (slow-down-robot robot)
           ()
  (slow-down body))


(defmethod (speed-up-robot robot)
           ()
  (speed-up body))


(defmethod (stable robot)
           ()
  (stable body supporting-legs))


(defmethod (stable_m robot)
           ()
  (stable-m body supporting-legs))


(defmethod (stable-without robot)
           (a-leg)
  (stable body (remove a-leg supporting-legs)))
```

```
defmethod (update-robot-status robot)
          ()
 (scanning self)
 (setf lift-flag t)
 (setf lift-able-legs (find-lift-able-legs self))
 (setf place-able-legs (find-place-able-legs self))
 (setf supporting-legs (find-supporting-legs self))
 (setf supporting-p-legs (find-supporting-p-legs self))
 (mapcar 'update-tkm-p supporting-p-legs)
 (if (check-tkm-limit-p self)
     (stop-motion body (check-tkm-limit-p self))
     (restore-motion body))
 (if (not (check-stability-p self))
     (modify-command-p body)
     (restore-command-p body))
 (calculate-motion body joy-command legs)
 (mapcar 'select-foothold place-able-legs)
 (mapcar 'update-tkm supporting-legs))
```

```
*********************************************************

  prolog interface functions

*********************************************************
```

```
defun at_tkm_limit()
 (at-tkm-limit asv))
```

```
defun do_recovery()
 (do-recovery asv))
```

```
defun execute_planned_motion()
 (execute-planned-motion asv))
```

```
defun graphical_display()
 (graphical-display asv))
```

```
defun has_more_tkm(leg1 leg2)
 (has-more-tkm asv leg1 leg2))
```

```
(defun inits()
  (init asv))


(defun leg_with_new_foothold()
  (leg-with-new-foothold asv))


(defun max_sm_leg(a-leg)
  (max-sm-leg asv a-leg))


(defun modify_command()
  (modify-command asv))


(defun read_joystick()
  (read-joystick asv))


(defun restore_command()
  (restore-command asv))


(defun send_decision(leg1 leg2 a-decision)
  (send-decision asv leg1 leg2 a-decision))


(defun smallest_tkm_leg()
  (smallest-tkm-leg asv))


(defun slow_down_robot()
  (slow-down-robot asv))


(defun speed_up_robot()
  (speed-up-robot asv))


(defun stable_p()
  (stable asv))


(defun stable_p_m()
  (stable_m asv))


(defun stable_without(a-leg)
  (stable-without asv a-leg))
```

```
defun update_robot_status()
 (update-robot-status asv))
```

```
;; -*- Mode: LISP; Syntax: Common-lisp; Package: LEG -*-

************************************************************

  sensor flavor definition

************************************************************

defflavor sensor(state owner)
          ()
 :initable-instance-variables)



************************************************************

  contact-sensor flavor definition

************************************************************

defflavor contact-sensor()
          (sensor)
 :initable-instance-variables)


defmethod (init contact-sensor)
          (leg-name)
 (setf state (sensing self)))


defmethod (contact-p sensor)
          ()
 state)


defmethod (sensing sensor)
          ()
 simulation purpose
 (setf state
       (let* ((leg-pos-wrt-body (leg-pos-wrt-body
                                  (leg-executor owner)))
              (leg-pos-wrt-earth
                (to-earth-transform (get-H1 owner) leg-pos-wrt-body))
              (x-y-pos (list (first leg-pos-wrt-earth)
                             (second leg-pos-wrt-earth)))
              (leg-height (third leg-pos-wrt-earth)))
         (if (< leg-height (+ 0.07 (third
                                     (terrain-point owner x-y-pos))))
             t
             nil))))
```

```
; -*- Mode: LISP; Syntax: Common-lisp; Package: BODY -*-


;*********************************************************

  stability-calculator flavor definition

;*********************************************************

defflavor stability-calculator(convex-hull-order
                               safety-margin
                               safety-margin-p
                               large-safety-margin
                               large-safety-margin-p
                               recovery-vector
                               recovery-vector-p
                               owner)
          ()
 :initable-instance-variables)


defmethod (init stability-calculator)
          ()
 (setf convex-hull-order '(leg2 leg4 leg6 leg5 leg3 leg1))
 (setf safety-margin 0.4)
 (setf safety-margin-p 0.2)
 (setf large-safety-margin 0.5)
 (setf large-safety-margin-p 0.4)
 (setf recovery-vector '(0 0 0))
 (setf recovery-vector-p '(0 0 0)))


defmethod (get-recovery-vector stability-calculator)
          ()
 recovery-vector)


defmethod (get-recovery-vector-p stability-calculator)
          ()
 recovery-vector-p)


defmethod (convert-to-recovery-vector stability-calculator)
          (stability-vector)
 (let ((sm (first stability-vector))
       (vect (second stability-vector)))
   (cond ((< sm 0)
          nil)
         ((< sm 0.1)
          (magvect (/ 1 sm) vect))
         (t
          (magvect (/ 0.1 (* sm sm)) vect)))))


defmethod (more-stable stability-calculator)
          (supporting-legs H leg1 leg2)
 (let ((stability1 (calculate-stability self
                                        (cons leg1 supporting-legs) H))
       (stability2 (calculate-stability self
                                        (cons leg2 supporting-legs) H)))
   (if (> stability1 stability2)
       t
```

```
        nil)))


(defmethod (stable-m stability-calculator)
; predict H <= H10
           (supporting-legs H)
  (let ((stability-vector
          (get-stability self supporting-legs H)))
    (cond ((>= (first stability-vector)
               large-safety-margin)
           t)
          (t
           (if (>= (first stability-vector) safety-margin)
               (setf recovery-vector
                     (convert-to-recovery-vector self stability-vector))
               (setf recovery-vector '(0 0 0)))
           nil))))


(defmethod (stable-p-m stability-calculator)
; present H <= H1
           (supporting-p-legs H)
  (let* ((stability-vector
          (get-stability self supporting-p-legs H))
         (st-margin (first stability-vector)))
    (cond ((>= st-margin
               large-safety-margin-p)
           t)
          (t
           (setf recovery-vector-p
                 (convert-to-recovery-vector self stability-vector))
;          (if (< st-margin safety-margin-p)
;              (print (list 'st-p st-margin)))
           nil))))


(defmethod (stable stability-calculator)
           (supporting-legs H10)
  (if (>= (calculate-stability self
                               supporting-legs H10)
          safety-margin)
      t
      nil))


(defmethod (stable-p stability-calculator)
           (supporting-p-legs H1)
  (if (>= (calculate-stability self
                               supporting-p-legs H1)
          safety-margin-p)
      t
      nil))


(defmethod (calculate-stability stability-calculator)
           (supporting-legs H)
  (first (get-stability self supporting-legs H)))
```

```
efmethod (get-stability stability-calculator)
         (supporting-legs H)
(if (>= (counting supporting-legs) 3)
    (measure-distance (center-of-gravity H)
                      (convex-hull-points
                        (supporting-points
                          (find-order self supporting-legs))))
    '(-100.0 (0 0 0))))


efun center-of-gravity (H)
center-of-body is represented wrt earth coordinate.
(let ((x (aref H 0 3))
      (y (aref H 1 3)))
  (list x y)))
center-of-body can be changed in future.


efun convex-hull-points (points)
point is a list(x y z).  For a time being, only x,y are used.
math lib : delete-list
(if (> (counting points) 3)
    (let* ((boundary-points (out-side points))
           (remaind (delete-list boundary-points points)))
      (cond (remaind (convex-hull-points boundary-points))
            (T       boundary-points)))
    points))  ; minimum points (3) are reached.



efmethod (find-order stability-calculator)
         (legs)
globals : leg-name
find ordered leg-names for calculating convex-hull-points
convex-hull-order (This has only ready position for leg names)
(let* ((ordered-legs nil))
  (empty-queue ordered-legs)
  (dolist (a-leg-name convex-hull-order)
    (dolist (a-leg legs)
      (if (equal a-leg-name (leg-name a-leg))
          (cons a-leg ordered-legs))))
   (reverse ordered-legs)))
          (enqueue ordered-legs a-leg))))
  ordered-legs))



efun find-slope (first-point second-point)
(let ((del-x (- (car  second-point) (car  first-point)))
      (del-y (- (cadr second-point) (cadr first-point))))
  (if (> (abs del-x) 0.0000001)
      (/ del-y del-x)
      nil)))




efun infinite-case (x a-line)
(list x
      (+ (* (car a-line) x) (cadr a-line))))
```

```
(defun intersection-point (a-line b-line)
; Returns list (x y).      Line is list (slope  crossing-point-of-axis).
  (cond ((null (car a-line)) (infinite-case (cadr a-line) b-line))
        ((null (car b-line)) (infinite-case (cadr b-line) a-line))
        (t  (normal-case a-line b-line))))



(defun in-side-of-convex-hull (center-point first-points second-points)
  (do* ((first-points  first-points  (cdr first-points))
        (second-points second-points (cdr second-points))
        (in-side-flag T))
       ((null first-points) in-side-flag)
    (if (test-out-side (car first-points) center-point (car second-points))
        (setf in-side-flag nil))))



(defun line (slope point)
  (if slope
      (list slope (- (second point) (* slope (first point))))
      (list slope (first point))))
; When slope is infinitive, return with x-axis crossing point instead of
; y-axis crossing point.



(defun measure-distance (center-point  convex-points)
; convex-points is a list of points
; point is a list (x y z).
  (let* ((first-points convex-points)
         (second-points (append (cdr convex-points)
                 (list (car first-points)))))
    (if (in-side-of-convex-hull center-point first-points second-points)
        (start-measure center-point first-points second-points)
        '(-10.0 (0 0 0)))))
                 ; center-of-gravity is out-side of support pattern



(defun normal-case (a-line b-line)
  (let* ((a1 (car a-line))
         (b1 (cadr a-line))
         (a2 (car b-line))
         (b2 (cadr b-line))
         (x  (/ (- b1 b2) (- a2 a1)))
         (y  (+ (* a1 x) b1)))
    (list x y)))



(defun out-side (points)
; this function does not change the order of points except deletion.
  (do* ((first-points points (cdr first-points))
        (second-points (reverse (cons (car points) (reverse (cdr points))))
                       (cdr second-points))
        (third-points  (reverse (cons (car second-points)
                                      (reverse (cdr second-points))))
                       (cdr third-points))
```

```
    (out-points nil out-points))
   ((null first-points)
    (let ((return-points nil))
      (empty-queue return-points)
      (dolist (a-point points)
        (if (member a-point out-points)
            (enqueue return-points a-point)))
      return-points))
  (if (test-out-side (car first-points) (car second-points)
                     (car third-points))
      (setf out-points (cons (car second-points) out-points)))))


efun point-distance (center-point first-point second-point)
returns distance and vector between cross-pt and center-pt
(let* ((slope1 (find-slope first-point second-point))
       (slope2 (right-angle slope1))
       (cross-pt (intersection-point (line slope1 first-point)
                                     (line slope2 center-point)))
       (del-vect (vectsub center-point cross-pt ))
       (distance (magnitude del-vect)))
  (list distance (list (first del-vect) (second del-vect) 0.0))))


efun right-angle (slope)
(cond ((null slope) 0.0)                ; infinitive input slope
      ((< (abs slope) 0.0000001) nil) ; zerop slope
      (t (/ (- 1) slope))))


efun start-measure (center-point first-points second-points)
(do* ((first-points first-points (cdr first-points))
      (second-points second-points (cdr second-points))
      (min-distance 10000.0 min-distance)   ; infinte dummy number 10000.0
      (min-direction nil) (dis-dir nil))
     ((null first-points)  (list min-distance min-direction))
  (setf dis-dir (point-distance center-point
                                (car first-points) (car second-points)))
  (cond ((< (first dis-dir) min-distance)
         (setf min-distance (first dis-dir))
         (setf min-direction (second dis-dir))))))


efun supporting-points (legs)
(let ((points))
  (empty-queue points)
  (do ((legs legs (cdr legs))
       (a-pos nil))
      ((null legs)
       points)
    (setf a-pos (leg-foothold (car legs)))
    (if a-pos
        (enqueue points a-pos)))))


efun test-out-side (first-point second-point third-point)
(let* ((a (- (cadr first-point) (cadr third-point)))
       (b (- (car  third-point) (car  first-point)))
       (c (- (+ (* a (car third-point)) (* b (cadr third-point))))))
```

```
        (decision (+ (* a (car  second-point))
                     (* b (cadr second-point))
                     c)))
    (if (>= decision 0.0)
        T
        nil)))
```

```
; -*- Mode: LISP; Syntax: Common-lisp; Package: BODY -*-

;*************************************************************

  support-plane-estimator flavor definition

;*************************************************************

defflavor support-plane-estimator(owner)
          ()
)


defmethod (init support-plane-estimator)
          ()
)


defmethod (get-plane support-plane-estimator)
          (legs)
 (let* ((footholds-for-estimation (get-footholds legs))
        (constants (get-constants footholds-for-estimation)))
     (make-plane-from-coefficient constants)))




;*************************************************************

  support-plane-estimator.get-plane

;*************************************************************


defun add-points (points)
 returns a list (number-of-points  sum-of-points).
 (do ((points points (cdr points))
      (i 0 (+ i 1))
      (sum-vect '(0 0 0)))
     ((null points) (list i sum-vect))
   (setf sum-vect (vectadd (car points) sum-vect))))


defun average-point (points)
 (let* ((num-&-sum-vect (add-points points))
        (number-of-points (first num-&-sum-vect))
        (sum-vect (second num-&-sum-vect)))
   (if (> number-of-points 0)
       (magvect (/ 1 number-of-points) sum-vect)
       (print "Error in finding average-point of estimate plane"))))


defun get-a0 (bar-point a1)
 (let* ((x-bar (first bar-point))
        (z-bar (third bar-point)))
   (- z-bar (* a1 x-bar))))
```

```
(defun get-a1 (points bar-point common-denominator)
; returns a1 which is sum in this function.
  (do* ((points points (cdr points))
        (sum 0)
        (x nil)  (x-bar (first bar-point))
        (z nil)  (z-bar (third bar-point)))
       ((null points) (/ sum common-denominator))
    (setf x (first (car points)))
    (setf z (third (car points)))
    (setf sum (+ sum (* (- x x-bar) (- z z-bar))))))


(defun get-a2 (points bar-point common-denominator)
; returns a2 which is sum in this function.
  (do* ((points points (cdr points))
        (sum 0)
        (x nil)  (x-bar (first  bar-point))
        (y nil)  (y-bar (second bar-point)))
       ((null points) (/ sum common-denominator))
    (setf x (first  (car points)))
    (setf y (second (car points)))
    (setf sum (+ sum (* (- x x-bar) (- y y-bar))))))


(defun get-a3 (bar-point a2)
  (let* ((x-bar (first  bar-point))
         (y-bar (second bar-point)))
    (- y-bar (* a2 x-bar))))


(defun get-a4 (points a0 a1 a2 a3)
  (let* ((number-of-points (counting points))
         (yr (get-yr points a2 a3))
         (zr (get-zr points a0 a1))
         (yr-bar (get-yr-bar yr number-of-points))
         (zr-bar (get-zr-bar zr number-of-points)))
    (do ((yr     yr     (cdr yr))
         (zr     zr     (cdr zr))
         (numerator 0) (a-yr 0) (a-zr 0)
         (denominator 0))
        ((null yr) (/ numerator denominator))
      (setf a-yr (first yr))
      (setf a-zr (first zr))
      (setf numerator   (+ numerator   (* (- a-yr yr-bar) (- a-zr zr-bar))))
      (setf denominator (+ denominator (* (- a-yr yr-bar) (- a-yr yr-bar)))))))


(defun get-common-denominator (points bar-point)
  (do* ((points points (cdr points))
        (sum 0)
        (x nil)
        (x-bar (first bar-point)))
       ((null points) sum)
    (setf x (first (car points)))
    (setf sum (+ sum (* (- x x-bar) (- x x-bar))))))
```

```
efun get-constants (points)
(let* ((bar-point (average-point points))
       (common-denominator (get-common-denominator points bar-point))
       (a1 (get-a1 points bar-point common-denominator))
       (a2 (get-a2 points bar-point common-denominator))
       (a0 (get-a0 bar-point a1))
       (a3 (get-a3 bar-point a2))
       (a4 (get-a4 points a0 a1 a2 a3)))
  (list a0 a1 a2 a3 a4)))



efun get-footholds (legs)
(do* ((legs legs (cdr legs))
      (footholds nil)
      (a-leg nil))
     ((null legs) footholds)
  (setf a-leg (car legs))
  (if (leg-foothold a-leg)
      (setf footholds (cons (leg-foothold a-leg) footholds)))))



efun get-yr (points a2 a3)
(do* ((points points (cdr points))
      (yr nil)
      (x  nil)
      (y  nil))
     ((null points) (reverse yr))
  (setf x (first  (car points)))
  (setf y (second (car points)))
  (setf yr (cons (- y a2 (* a3 x)) yr))))



efun get-yr-bar (yr number-of-points)
(do ((yr yr (cdr yr))
     (yr-bar 0))
    ((null yr) (/ yr-bar number-of-points))
  (setf yr-bar (+ yr-bar (first yr)))))



efun get-zr (points a0 a1)
(do* ((points points (cdr points))
      (zr nil)
      (x  nil)
      (z  nil))
     ((null points) (reverse zr))
  (setf x (first  (car points)))
  (setf z (third  (car points)))
  (setf zr (cons (- z a0 (* a1 x)) zr))))




efun get-zr-bar (zr number-of-points)
(do ((zr zr (cdr zr))
     (zr-bar 0))
    ((null zr) (/ zr-bar number-of-points))
  (setf zr-bar (+ zr-bar (first zr)))))
```

```
(defun make-plane-from-coefficient (constants)
  (let* ((a0 (first  constants))
         (a1 (second constants))
         (a2 (third  constants))
         (a3 (fourth  constants))
         (a4 (fifth  constants))
         (a  (- (* a4 a3) a1))
         (b  (- a4))
         (c  1)
         (d  (- (* a2 a4) a0))
         (unit-normal (normalize-vector (list a b c)))
         (dis (/ d (magnitude (list a b c)))))
    (list unit-normal dis)))
```

```
;; -*- Mode: LISP; Syntax: Common-lisp; Package: BODY -*-

*******************************************************

  terrain-regulator flavor definition

*******************************************************

defflavor terrain-regulator(body-rotate-rate-x body-rotate-rate-y
                            body-trans-rate-z  old-body-rotate-rate-x
                            old-body-rotate-rate-y old-body-trans-rate-z
                            gain min-height max-height
                            eta1 eta2 min-eta max-eta)
          (regulator)
 :initable-instance-variables)



defmethod (init terrain-regulator)
          ()
 (setf gain 5)

 (setf min-eta 0.0000001)               ; 0   degree
 (setf max-eta 0.4363)                  ; 25  degrees
 (setf min-height 4.4)                  ; 4.4 feet
 (setf max-height 5.4)                  ; 5.4 feet
 (setf eta1 min-eta)                    ; 0   degree
 (setf eta2 0.5236)                     ; 30  degree

 (setf body-rotate-rate-x 0.0)
 (setf body-rotate-rate-y 0.0)
 (setf body-trans-rate-z 0.0)
 (list body-rotate-rate-x body-rotate-rate-y body-trans-rate-z))



defmethod (do-terrain-regulation terrain-regulator)
          (k-gamma-delta-height)
 k-gamma-delta-height is ((k.x k.y k.z) gamma delta-height).
 (let* ((k     (first  k-gamma-delta-height))
        (gamma (second k-gamma-delta-height))
        (delta-height (third k-gamma-delta-height))
        (body-rotate-rate-x-n (* gain (first  k) gamma))
        (body-rotate-rate-y-n (* gain (second k) gamma))
        (body-trans-rate-z-n  (* gain delta-height)))
   (setf body-rotate-rate-x
         (limitor self
           (filter self body-rotate-rate-x-n body-rotate-rate-x)
           0.1))
   (setf body-rotate-rate-y
         (limitor self
           (filter self body-rotate-rate-y-n  body-rotate-rate-y)
           0.1))
   (setf body-trans-rate-z
         (limitor self
           (filter self body-trans-rate-z-n body-trans-rate-z)
           1)))
 (list body-rotate-rate-x body-rotate-rate-y body-trans-rate-z))



defmethod (eta-function terrain-regulator)
```

```
          (eta)
  (let ((slope (/ (- max-eta min-eta) (- eta2 eta1))))
    (+ min-eta (* slope (- eta eta1)))))



(defmethod (get-k-gamma-by-slope terrain-regulator)
          (plane H)
  (let* ((plane-rpt-body (plane-transform plane H))
         (height (cadr plane-rpt-body))
         (eta (arc-cos (third (car plane))))
         (k-gamma-desired-height nil))
    (setf k-gamma-desired-height
          (cond ((< eta eta1) (low-slope self plane))
                ((< eta eta2) (mid-slope self eta plane H))
                (T            (high-slope self plane H))))
    (list (first  k-gamma-desired-height)
          (second k-gamma-desired-height)
          (- (third k-gamma-desired-height) height))))



(defmethod (height-function terrain-regulator)
          (eta)
  (let ((slope (/ (- max-height min-height) (- eta2 eta1))))
    (- max-height (* slope (- eta eta1)))))



(defmethod (high-slope terrain-regulator)
          (plane H)
  (let* ((plane-unit-normal (first plane))
         (a (first  plane-unit-normal))
         (b (second plane-unit-normal))
         (m (sqrt (+ (* a a) (* b b))))
         (desired-eta max-eta)
         (desired-height min-height)
         (desired-body-plane (list (list (* (/ a m) (sin desired-eta))
                                         (* (/ b m) (sin desired-eta))
                                         (cos desired-eta)) 0.0))
         (desired-body-plane-in-body (plane-transform desired-body-plane H))
         (unit-normal-body-plane (first desired-body-plane-in-body))
         (a1 (first  unit-normal-body-plane))
         (b1 (second unit-normal-body-plane))
         (c1 (third  unit-normal-body-plane))
         (m1 (sqrt (+ (* a1 a1) (* b1 b1))))
         (k  (if (= m1 0)
                 (list 0 0 0)
                 (list (/ (- b1) m1) (/ a1 m1) 0)))
         (gamma (arc-cos c1)))
    (list k gamma desired-height)))



(defmethod (limitor terrain-regulator)
          (vel max-vel)
  (if (>= (abs vel) max-vel)
      (if (> vel 0)
          max-vel
          (- max-vel))
      vel))
```

```
defmethod (low-slope terrain-regulator)
          (plane)
 (let* ((unit-normal (first plane))
        (a (first  unit-normal))
        (b (second unit-normal))
        (c (third  unit-normal))
        (m (sqrt (+ (* a a) (* b b))))
        (k.a nil)
        (k.b nil)
        (gamma (arc-cos c))
        (desired-height max-height))
   (if (= m 0.0)
       (setf k.a 0.0          k.b 0.0)
       (setf k.a (/ (- b) m)  k.b (/ a m)))
   (list (list k.a k.b 0.0) gamma desired-height)))



defmethod (mid-slope terrain-regulator)
          (eta plane H)
 (let* ((plane-unit-normal (first plane))
        (a (first  plane-unit-normal))
        (b (second plane-unit-normal))
        (m (sqrt (+ (* a a) (* b b))))
        (desired-eta (eta-function self eta))
        (desired-height (height-function self eta))
        (desired-body-plane (list (list (* (/ a m) (sin desired-eta))
                                        (* (/ b m) (sin desired-eta))
                                        (cos desired-eta)) 0.0))
        (desired-body-plane-in-body (plane-transform desired-body-plane H))
        (unit-normal-body-plane (first desired-body-plane-in-body))
        (a1 (first  unit-normal-body-plane))
        (b1 (second unit-normal-body-plane))
        (c1 (third  unit-normal-body-plane))
        (m1 (sqrt (+ (* a1 a1) (* b1 b1))))
        (k  (if (= m1 0)
                (list 0 0 0)
                (list (/ (- b1) m1) (/ a1 m1) 0)))
        (gamma (arc-cos c1)))
   (list k gamma desired-height)))



defmethod (regulate terrain-regulator)
          (estimated-support-plane H)
 (let ((k-gamma (get-k-gamma-by-slope self estimated-support-plane H)))
   (do-terrain-regulation self k-gamma)))



defmethod (restore terrain-regulator)
          ()
 (setf body-rotate-rate-x old-body-rotate-rate-x)
 (setf body-rotate-rate-y old-body-rotate-rate-y)
 (setf body-trans-rate-z  old-body-trans-rate-z)
 (list body-rotate-rate-x body-rotate-rate-y body-trans-rate-z))



defmethod (save terrain-regulator)
          ()
 (setf old-body-rotate-rate-x body-rotate-rate-x)
```

```
  (setf old-body-rotate-rate-y body-rotate-rate-y)
  (setf old-body-trans-rate-z  body-trans-rate-z)
  )
```

```
; -*- Mode: LISP; Syntax: Common-lisp; Package: LEG -*-

;*******************************************************

 tkm-calculator flavor definition

;*******************************************************

defflavor tkm-calculator(working-volume owner)
          ()
 :initable-instance-variables)


defmethod (init tkm-calculator)
          (leg-name)
 (cond ((equal leg-name 'leg1)
        (setf working-volume
              '((((0 0 1) 3.316)  ((1 0 0) -8.0832) ((0 0.9397  0.3420) -2.569))
                (((0 0 1) 5.7313) ((1 0 0) -3.4167) ((0 0.9397 -0.3420) -2.569)))))
       ((equal leg-name 'leg2)
        (setf working-volume
              '((((0 0 1) 3.316)  ((1 0 0) -8.0832) ((0 0.9397  0.3420)  2.569))
                (((0 0 1) 5.7313) ((1 0 0) -3.4167) ((0 0.9397 -0.3420)  2.569)))))
       ((equal leg-name 'leg3)
        (setf working-volume
              '((((0 0 1) 3.316)  ((1 0 0) -2.2915) ((0 0.9397  0.3420) -2.569))
                (((0 0 1) 5.7313) ((1 0 0)  2.2915) ((0 0.9397 -0.3420) -2.569)))))
       ((equal leg-name 'leg4)
        (setf working-volume
              '((((0 0 1) 3.316)  ((1 0 0) -2.2915) ((0 0.9397  0.3420)  2.569))
                (((0 0 1) 5.7313) ((1 0 0)  2.2915) ((0 0.9397 -0.3420)  2.569)))))
       ((equal leg-name 'leg5)
        (setf working-volume
              '((((0 0 1) 3.316)  ((1 0 0)  3.3332) ((0 0.9397  0.3420) -2.569))
                (((0 0 1) 5.7313) ((1 0 0)  7.8332) ((0 0.9397 -0.3420) -2.569)))))
       ((equal leg-name 'leg6)
        (setf working-volume
              '((((0 0 1) 3.316)  ((1 0 0)  3.3332) ((0 0.9397  0.3420)  2.569))
                (((0 0 1) 5.7313) ((1 0 0)  7.8332) ((0 0.9397 -0.3420)  2.569)))))
       )
 )


defmethod (find-tkm tkm-calculator)
          (a-foothold body-trans-rate body-rotate-rate)
 a-foothold is based on body coordinate
 returns tkm
 (let* ((leg-vel-rpt-body
          (get-leg-velocity
            a-foothold body-trans-rate body-rotate-rate)))
   (get-tkm a-foothold leg-vel-rpt-body working-volume)))


defun get-distance (planes velocity leg-position)
 global function : plane-distance
 before start, make one plane list
 (do ((planes (append (first planes) (second planes)) (cdr planes))
      (a-tkm nil)
      (min-tkm 10000))
     ((null planes) min-tkm)
```

```
    (setf a-tkm (plane-distance (car planes) velocity leg-position))
    (if a-tkm
        (if (and (> a-tkm 0) (> min-tkm a-tkm))
            (setf min-tkm a-tkm)))))



(defun get-leg-velocity (pos-rpt-body body-trans-rate body-rotate-rate)
; returns leg-velocity-wrt-body
;         = - ( body-trans-rate + body-rotate-rate X pos-rpt-body )
  (vectsub '(0 0 0)
           (vectadd body-trans-rate
                    (crossprod body-rotate-rate pos-rpt-body))))



(defun get-tkm (leg-pos-rpt-body velocity working-volume)
; global function : magnitude
; outside w.v returns nil.  If speed is near 0, then returns 1000.0.
  (if (in-side-volume leg-pos-rpt-body working-volume)
      (let ((speed (magnitude velocity)))
        (if (< speed 1/1000)
            1000.0
            (/ (get-distance working-volume velocity leg-pos-rpt-body) speed)))
      nil))



(defun in-side-volume (position planes)
; planes ((up front left) (back right bottom))
  (let* ((positive-planes (first planes))
         (negative-planes (second planes))
         (inside-flag T))
    (dolist (a-plane positive-planes)
      (if (>= (plane-normal-distance a-plane position) 0)
          (setf inside-flag nil)))
    (dolist (a-plane negative-planes)
      (if (<= (plane-normal-distance a-plane position) 0)
          (setf inside-flag nil)))
    inside-flag))
```

```
;; -*- Mode: LISP; Syntax: Common-lisp; Package: VISION -*-


defflavor vision-system (owner)
          ()
 :initable-instance-variables)


defmethod (init vision-system)
          ()
 )


defmethod (scanning vision-system)
          ()
 )


defmethod (permitted-cell vision-system)
          (t-cell)
 (permitted-cell graph-terrain t-cell))



defmethod (terrain-point vision-system)
          (t-cell)
 (terrain-point graph-terrain t-cell))
```

```
 13
      6.625   0.0     3.0)
      6.625   0.0     1.08)
      6.625  -2.0     1.08)
     -6.625  -2.0     1.08)
     -6.625   2.0     1.08)
      6.625   2.0     1.08)
      6.625   0.9    -3.1)
      6.625  -0.9    -3.1)
     -6.625  -0.9    -3.1)
     -6.625   0.9    -3.1)
      0.0     0.0     0.0)
      0.0     0.0     0.0)
      0.0     0.0     0.0)
      0.0     0.0     0.0)
      0.0     0.0     0.0)
      0.0     0.0     0.0)
      0.0     0.0     0.0)
      0.0     0.0     0.0)
      0.0     0.0     0.0)
      0.0     0.0     0.0)
      0.0     0.0     0.0)
      0.0     0.0     0.0)
      0.0     0.0     0.0)
      0.0     0.0     0.0)
      0.0     0.0     0.0)
      0.0     0.0     0.0)
      0.0     0.0     0.0)
 1 2)
 3 4 5 6 3)
 6 7)
 5 10)
 4 9)
 3 8)
 8 7 10 9 8)
 11 12 13)
 14 15 16)
 17 18 19)
 20 21 22)
 23 24 25)
 26 27 28)
```

Distribution List

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Library, Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Center for Naval Analyses 4401 Ford Anenue Alexandria, Virginia 22302-0268	1
Chief of Naval Operations Director, Information Systems (OP-945) Navy Department Washington, D.C. 20350-2000	1
Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1
Department Chairman, Code 62 Department of Electronics and Computer Engineering Naval Postgraduate School Monterey, CA 93943	1
Professor Robert B. McGhee, Code 52Mz. Department of Computer Science Naval Postgraduate School Monterey, CA 93943	12
Professor Neil C. Rowe, Code 52Rp Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
Professor Michael J. Zyda, Code 52Zk Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
Professor Roberto Cristi, Code 62Cx Department of Electronic and Computer Engineering Naval Postgraduate School Monterey, CA 93943	1

Department Chairman, Code 69 1
Department of Mechanical Engineering
Naval Postgraduate School
Monterey, CA 93943

Professor D. L. Smith, Code 69Sm 1
Department of Mechanical Engineering
Naval Postgraduate School
Monterey, CA 93943

Major Robert Richbourg 1
USMA
Office of Artificial Intelligence Analysis and Evaluation
Attention: MADN-B
West Point, New York 10996

Professor Kenneth J. Waldron 1
Department of Mechanical Engineering
Ohio State University
206 West 18th Avenue
Columbus, Ohio 43210

Professor C. A. Klein 1
Department of Electrical Engineering
Ohio State University
2015 Neil Avenue
Columbus, Ohio 43210

Professor D. E. Dain 1
Department of Electrical Engineering
Ohio State University
2015 Neil Avenue
Columbus, Ohio 43210

Doctor William Isler 1
DARPA/ISTO
1400 Wilson Boulevard
Arlington, Virginia 22209

Doctor Robert Rosenfeld 1
DARPA/ISTO
1400 Wilson Boulevard
Arlington, Virginia 22209

Professor A. J. Healey 1
Department of Mechanical Engineering
Naval Postgraduate School
Monterey, CA 93943